AMORE: Adaptive Multi-Output Operator Network for Stiff Chemical Kinetics

Kamaljyoti Nath [a], **Additi Pandey** [a], **Bryan T. Susi** [b], **Hessam Babaee** [c], **George Em Karniadakis** [a]

[a] Division of Applied Mathematics, Brown University, Providence, RI, USA
[b] Applied Research Associates, Inc., Raleigh, NC, USA
[c] Department of Mechanical Engineering and Materials Science, University of Pittsburgh, Pittsburgh, PA, USA

Abstract

Time integration of stiff systems is a primary source of computational cost in combustion, hypersonics, and other reactive transport systems. This stiffness can introduce time scales significantly smaller than those associated with other physical processes, requiring extremely small time steps in explicit schemes or computationally intensive implicit methods. Consequently, strategies to alleviate challenges posed by stiffness are important. While neural operators (DeepONets) can act as surrogates for stiff kinetics, a reliable operator learning strategy is required to appropriately account for differences in the error between output variables and samples. Here, we develop AMORE, Adaptive Multi-Output Operator Network, a framework comprising an operator capable of predicting multiple outputs and adaptive loss functions ensuring reliable operator learning. The operator predicts all thermochemical states from given initial conditions. We propose two adaptive loss functions within the framework, considering each state variable's and sample's error to penalize the loss function. We designed the trunk to automatically satisfy Partition of Unity. To enforce unity mass-fraction constraint exactly, we propose an invertible analytical map that transforms the n-dimensional species mass-fraction vector into an $(n-1)$-dimensional space, where DeepONet training is performed. We consider two-step training for DeepONet for multiple outputs and extend adaptive loss functions for trunk and branch training. We demonstrate the efficacy and applicability of our models through two examples: the syngas (12 states) and GRI-Mech 3.0 (24 active states out of 54). The proposed DeepONet will be a backbone for future CFD studies to accelerate turbulent combustion simulations. AMORE is a general framework, and here, in addition to DeepONet, we also demonstrate it for FNO.

***K*eywords** DeepONet · Stiff dynamical system · Partition of Unity · Adaptive loss function · Kolmogorov Arnold networks (KANs) · Fourier Neural Operator (FNO)

1 Introduction

Chemical kinetics plays a key role spanning various scientific disciplines, extending beyond combustion, hypersonics, environmental engineering, chemical engineering, and biomedicine. In chemical kinetics, the ordinary differential equations (ODEs) representing the net production rate of the species involved in the reaction are stiff in nature because of the presence of widely varying reaction rates. This issue of stiffness becomes particularly challenging in turbulent combustion simulations, where chemical kinetics must be solved concurrently with the balance of mass, momentum, and energy conservation. In reactive flow simulations, the shortest time scales introduced by chemical reactions can be orders of magnitude smaller than those associated with key physical processes, such as convection and diffusion, collectively known as hydrodynamic time scales. This disparity becomes particularly challenging in high-fidelity turbulent combustion simulations, such as large eddy simulations (LES) or direct numerical simulations (DNS). In these simulations, the spatial resolution requirements are driven by the need to resolve the smallest turbulent length scales and/or flame thickness, resulting in extremely large computational grids. Under these conditions, even for a moderate number of chemical species, fully implicit time integration becomes computationally infeasible. Consequently, semi-implicit or operator-splitting schemes are often employed, wherein non-reactive terms are integrated explicitly, while the stiff chemical source terms are treated implicitly. However, implicit integration of chemical kinetics is itself highly expensive for direct solvers; the computational cost scales as $O(n^3)$, where n is the number of species. Given the large number of grid points required to resolve turbulent and flame structures, the overall computational cost remains a principal bottleneck in high-fidelity turbulent combustion simulations.

Corresponding author: G. E. Karniadakis (george_karniadakis@brown.edu)
E-mail addressees: kamaljyoti_nath@brown.edu (K. Nath), additi_pandey@brown.edu (A. Pandey), bsusi@ara.com (B. T. Susi), h.babaee@pitt.edu (H. Babaee), george_karniadakis@brown.edu (G. E. Karniadakis)

Consequently, alleviating the computational cost associated with chemical stiffness has rightfully merited substantial attention in the turbulent combustion literature, resulting in the development of numerous approaches that span a spectrum of trade-offs between *generalizability*, *accuracy*, and *computational cost*. A large subset of these methods is rooted in the principle of timescale separation, wherein fast species, subspaces, or manifolds are distinguished from their slower counterparts. Some well-known techniques to address stiffness in reactive systems include the quasi-steady state approximation (QSSA), partial equilibrium (PE), intrinsic low-dimensional manifold (ILDM) [1], reaction–diffusion manifold (REDIM) [2], computational singular perturbation (CSP) [3], slow invariant manifold (SIM) and approximate slow invariant manifold (ASIM). Among these, QSSA and PE are the most computationally efficient techniques. However, they require stringent equilibrium assumptions and expert-driven identification of fast species and reactions, which limits their generalizability and can compromise accuracy. The other techniques such as ILDM, REDIM, CSP, SIM, and ASIM, offer greater generalizability and accuracy by identifying slow and fast subspaces or manifolds; however, they require Jacobian evaluations, manifold construction, or numerical optimization, making them considerably more complex and computationally demanding than QSSA and PE. Recently, an on-the-fly reduced-order modeling approach was introduced to mitigate chemical stiffness [4]. Using a matrix cross-approximation algorithm, this method solves the species transport equations on a low-rank matrix manifold. It demonstrates strong generalizability and high accuracy, and reduces computational cost for the cases studied by at least an order-of-magnitude.

With the advancement of machine learning methods and techniques, there has been a noticeable increase in the use of artificial neural networks in place of classical numerical methods to solve a system of stiff differential equations due to their computational efficiency when dealing with large and complex systems and their accuracy in approximations. Moreover, the flexibility offered by neural networks, in terms of the model architecture, hyperparameters, and use of varied activation and loss functions, makes them adaptable for modeling complex systems. Sharma et al. [5] presented an artificial neural network-based model to approximate the stiff chemical kinetics and its integration into the computational fluid dynamics (CFD) code. They also enforce the conservation of the mass constraint via the softmax function on the species' predicted mass fractions. However, for each t^{th} time step input data, the model predicts the $(t+1)^{\text{th}}$ time step. Brown et al. [6] employed parallel ResNets, in which a separate ResNet is considered for each state variable with input dimension of $M+2$ corresponding to temperature, M species, and time increment δt. De Florio et al. [7] and Frankel et al. [8] employed physics-informed neural network (PINN) [9] based methodologies using the extreme theory of functional connections (X-TFC) [10] to solve stiff chemical kinetics problems. Stiff PINNs by Ji et al. [11] attempted to solve stiff ODEs using PINNs via a QSSA. The authors also highlighted that, for a complex chemical system, manually deriving the QSSA formula cannot only be time-consuming but also requires a thorough understanding of the process. Furthermore, mild stiffness may still persist in the reduced system. In general, although PINN-based methods can predict the dynamics of stiff chemical systems, they need to be trained for any change in the given conditions, such as the initial conditions.

Neural operator surrogates involve *offline* training costs, but their *online* (inference) costs are much smaller than implicit solvers. The advantage of using a neural operator is that once it has been trained offline, it can be used for fast online predictions over a new input dataset. Venturi and Casey [12] proposed flexDeepONet, a DeepONet [13] based architecture and employed it to analyze combustion chemistry in an isobaric reactor, where each thermodynamic state variable has its own branch and trunk network. The architecture prepends an additional pre-transformation network that uses branch input to transform the trunk input before they enter the trunk network. This allows discovery of a moving frame of reference, allowing multiple scenarios to be compressed into a lower number of modes. Goswami et al. [14] proposed different variants of DeepONet (e.g., DeepONet, PoU-DeepONet, and autoencoder-based DeepONet) for different stiff chemical kinetics problems such as ROBER, POLLU, pure kinetic syngas, and turbulent flame syngas problems. The authors considered different DeepONets for different species for ROBER and POLLU problems, with only one input at a time. Furthermore, the authors did not consider the dynamics and stiffness of the problem. While in the syngas problems, the DeepONet can predict multiple output variables, the training and predictions are considered at times $t+250\Delta t, t+500\Delta t, t+750\Delta t$ and $t+1000\Delta t$, with $\Delta t = 10^{-8}$ s. With such a high time stepping, there is a high possibility that the model does not take the stiffness (i.e., the peaks in the dynamics) into consideration. Furthermore, the authors have also confirmed that the auto-regressive prediction error reported in the paper for the syngas pure kinetics problem is not predicted using an auto-regressive prediction. In the present study, we have shown that our vanilla DeepONet and DeepOKAN can achieve better accuracy. Moreover, with the proposed adaptive loss functions and two-step training, we obtained even better and more reliable accuracy. Moreover, for the auto-regressive prediction, our results show that the 90 percentile of the samples has less than 5% error when predicted using an auto-regressive manner for the complete time series.

Kumar and Echekki [15] introduced React-DeepONet, where a subset of thermochemical scalars is evolved and others are reconstructed using ANNs. The model takes as input this subset, called representative scalars ϕ, at time t and a time increment δt and predicts them at $t+\delta t$. This model includes a pre-transformation network to transform the trunk input and a parameter input sub-network to encode parametric variations. In addition to the initial training, a refined training paradigm is adopted. A latent space React-DeepONet is also introduced for dimensionality reduction of representative scalars using an autoencoder to yield latent variables on which React-DeepONet is implemented. Weng et al. [16] implemented a Fourier neural operator (FNO) to model the stiff chemical kinetics of species via the Box-Cox transformation [17]. They introduce additional constraints, such as the conservation of mass and elements in the loss function, along with a balanced data loss function (based on

focal loss). The balanced loss function updates adaptively for every data sample, taking constant values for weights depending on the variation of mass fractions between adjacent time steps.

For efficient integration with a CFD solver, we require the neural operator to accurately predict the kinetics, ensuring that the stiffness is taken into account. Moreover, for computational efficiency and scalability, it is desirable that a single neural operator should be able to predict all thermo-chemical state variables, taking into account the difference in error between the state variables. To this end, we propose a framework called **A**daptive **M**ulti-output **O**perato**r** n**e**twork (AMORE) by considering a Deep Operator Network, popularly known as DeepONet, and its variants to accurately predict the dynamics of the stiff chemical system and adaptive loss functions to ensure reliable operator learning. One of the major advantages of using neural operators (e.g., DeepONet) that are a functional approximation model over a fully connected neural network (FCNN) that are function approximation model is that it can predict the dynamics of the output variables from the initial conditions, unlike one-step prediction in the case of an FCNN. DeepONet is a neural operator, proposed by Lu et al. [13], wherein the two one-hidden-layered networks of Chen and Chen [18] are replaced by two deep neural networks. These two networks are known as the branch and the trunk network of DeepONet, which aims to map an infinite-dimensional input function ($u \in \boldsymbol{U}$) to another infinite-dimensional output function ($v \in \boldsymbol{V}$) in the Banach space. The input function is taken by the branch network, which captures its characteristics, while the spatio-temporal coordinates where the output needs to be predicted are given as input into the trunk network. The output of the DeepONet is given by a dot product of the output of the branch and trunk networks. We can assume the trunk output as the universal basis function and the branch output as the coefficient for these basis functions for a particular input function. DeepONet has been successfully applied in modeling many complex problems [19–25], demonstrating its potential to act as a surrogate model to diverse fields of problems. One of the advantages of DeepONet is that it can be applied to simulated data, experimental data, or both [26], and it can predict a continuous output. Moreover, there is no restriction on the type of network that is considered in either the branch and trunk networks. With the recent proposal of Kolmogorov Arnold networks (KANs) [27], researchers have considered different variants of KAN as a choice of network architecture for the trunk and branch network [28, 29]. Recently Lee and Shin [30] proposed a novel two-step training strategy for DeepONet, where the trunk and the branch are trained in a sequential manner rather than being trained together. The method involves QR decomposition of the trunk output, making the basis function orthogonal and more expressive.

The present study aims to accurately model the dynamics of stiff chemical kinetics via the AMORE framework. Through this framework, we propose two different types of DeepONet architectures, using ResNet and KAN as choices of network architecture for the branch and trunk networks. To improve the accuracy of the prediction, we propose two adaptive loss functions within the framework. Furthermore, we consider the two-step training of DeepONet. In the next section, Section 1.1, we discuss the main contribution of the present study. The rest of the paper is structured as follows: Section 2, gives the overview of the problem we consider in the present study as well as the framework we propose, along with a discussion of the objective and importance of the present study. The methodologies employed to solve the problem are discussed in Section 3. In particular, we consider variants of DeepONet. This section includes a detailed discussion of the network architecture and implementation of these DeepONets, along with training strategies and various improvements needed to obtain good accuracy, including the proposed adaptive loss functions. The applicability and accuracy of the methods are evaluated by considering computational experiments. We have included two computational examples in Sections 4 and 5 along with a detailed comparative study. The proposed AMORE framework is a general framework. In Section 6, we have implemented and demonstrated it with FNO. The computational cost is discussed in Section 7. The summary and conclusion of the study are presented in Section 8 along with a discussion on the future scope of work.

1.1 Main contribution

In this work, we propose a neural operator framework, specifically DeepONets, to accurately predict the dynamics of stiff systems, particularly of stiff chemical systems described by the system of stiff ODEs. We call this framework adaptive multi-output operator network (AMORE) for stiff chemical kinetics. In Fig. 1, we have illustrated the AMORE framework, which consists of an operator network that is capable of predicting multiple outputs and adaptive loss functions. Since stiff chemical systems have multiple state variables (e.g., species' mass fractions and temperature), it is desired, and therefore, we propose, to consider a single DeepONet to predict all the state variables instead of considering separate DeepONets for each state variable. This configuration will help in the scalability of the proposed method for chemical systems involving a large number of state variables. We have trained our proposed DeepONet with the AMORE scheme. This framework will serve as a cornerstone for future studies on integrating neural operators with CFD solvers to improve the computational efficiency of the coupled models. We summarize the main contributions through our proposed AMORE framework as follows:

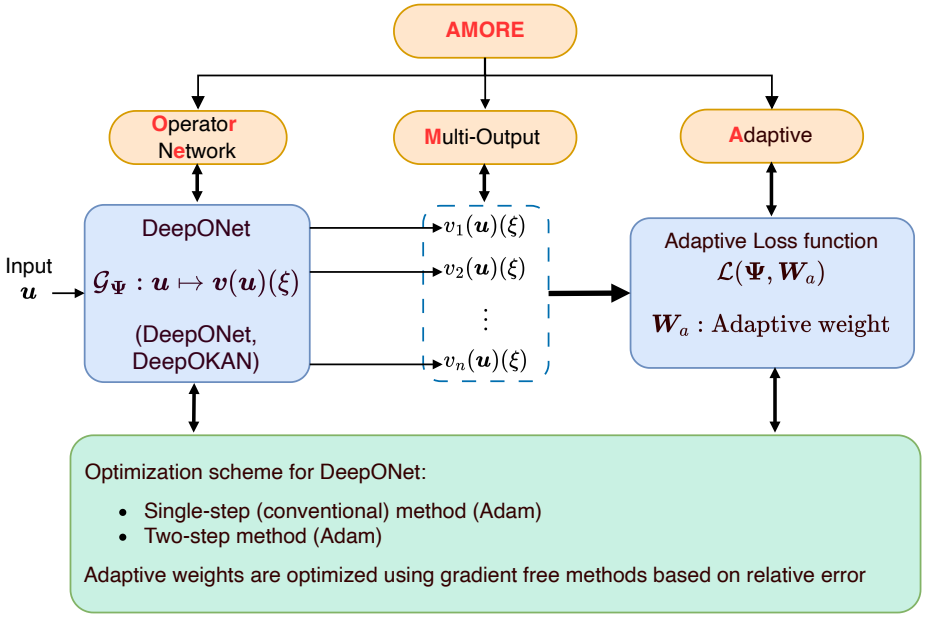


Fig. 1: **AMORE:** A schematic diagram of the proposed **A**daptive **M**ulti-output **O**perato**r** n**e**tworks (AMORE) framework. It consists of an operator network that is capable of predicting multiple outputs and adaptive loss functions. In the present study, we consider Deep Operator Network (DeepONet) with two different architectures as the choice of the operator network, the DeepONet and DeepOKAN. We propose two adaptive loss functions that are updated using a gradient-free method based on the relative error between the state variables and samples.

- **DeepONet architecture**: We propose two DeepONet architectures to predict the dynamics of the multiple state variables of stiff chemical systems from given initial conditions of these variables. These architectures consist of a single trunk network and a branch network. The first proposed architecture consists of ResNet as both the branch and trunk networks, while the second proposed architecture consists of a ResNet for the branch network and a KAN as the trunk network. We shall refer to them as DeepONet and DeepOKAN, respectively.
  - **Partition of Unity**: We design our trunk network such that it automatically satisfies the partition of unity (PoU) constraint in the trunk network.
  - **Conservation of mass**: We know that the total mass of the reactants and the products at any instant is the same. Hence, we consider a loss function due to conservation of mass (CoM) in terms of mass fraction along with the data loss. The sum of the mass fractions of all the species is one at any given time, and we consider it as an additional penalty in the loss function.
  - **Adaptive loss functions**: To improve prediction accuracy and efficient operator learning, we propose two gradient-free adaptive loss functions. The basic idea behind these loss functions is that the state variables and the samples with higher error need to be penalized more in the loss function.
- **Two-step training of DeepONet**: We consider the two-step training of DeepONet to predict the dynamics of the multiple state variables of stiff chemical systems from given initial conditions of these variables with one trunk but different basis functions for each output variable, and one branch but different coefficients for each output variable.
  - **Adaptive loss functions**: In the two-step training process, we extend the two adaptive loss functions in training both the trunk and the branch networks with the appropriate modifications.
- **Mass conserving DeepONet**: The loss function corresponding to the CoM constraint does not guarantee the conservation of mass in terms of the mass fraction of the species and depends on the penalty considered in the loss function. Consequently, we propose a mass-conserving neural operator that automatically satisfies the conservation of mass constraint at each instant for all the participating species whose sum of mass fractions is one.
- **AMORE implementation in FNO**: The proposed AMORE framework is a general framework. It can be used with other neural operators as well. To demonstrate this, we have implemented the AMORE framework with FNO.

Together, these contributions result in a remarkable improvement in prediction accuracy, surpassing the current state of the art. In the following sections, we will discuss the formulation of the framework, DeepONets, their detailed implementation, along with a discussion of each of the aforementioned contributions.

2 Problem statement

For demonstration purposes, we consider adiabatic, constant-pressure zero-dimensional reactions of ideal gases described by the following system of nonlinear ordinary differential equations:

$$\frac{dy_k}{dt} = f_{y_k}(\boldsymbol{y}, T) = \frac{W_k}{\rho} \sum_{j=1}^{n_r} \nu_{kj} \boldsymbol{Q}_j, \quad \boldsymbol{y}(0) = \boldsymbol{y}_0, \tag{1a}$$

$$\frac{dT}{dt} = f_T(\boldsymbol{y}, T) = -\frac{1}{c_p} \sum_{k=1}^{n} h_k f_{y_k}, \quad T(0) = T_0, \tag{1b}$$

where ρ is the gas mixture density, n_r is the number of reactions, c_p is the heat specific capacity of the gas mixture, W_k and h_k are the molecular weight and enthalpy of species k, respectively. Moreover, ν_{kj} is the net molar stoichiometric coefficient of species k and reaction j, and $\boldsymbol{Q}_j$ is the progress rate of reaction j.

Through the present study, our objective is to develop a comprehensive framework of neural operators that efficiently and accurately models a stiff ODE system governing a chemical-kinetics reaction mechanism (Eq. (1)). Once the neural operator is trained offline, it can be used as a surrogate for the time integration of Eq. (1). The neural operator maps the initial condition of the stiff ODEs (Eq. (1)) to the dynamics of the state variables, i.e., temperature and the species' mass fractions. Thus, the operator learning can be mathematically written as,

$$\mathcal{G}_{\boldsymbol{\Psi}} : \boldsymbol{Y}_0 \mapsto \boldsymbol{Y}(\boldsymbol{Y}_0)(t), \tag{2}$$

where $\boldsymbol{Y}_0 = [T(0), y_1(0), y_2(0), \cdots, y_n(0)]^\top$ is the initial condition corresponding to the temperature and the species' mass fractions. The output is the transient evolution of temperature and the species' mass fractions.

In Fig. 2, we have shown a schematic of the prediction using DeepONet for the problem we have considered. We have considered a system with computational time domain $[t_0, t_{\text{Final}}]$ with the time step for the chemical kinetics Δt. We assume the time step for the other associated physical process (e.g., CFD or a PINN model for mass, momentum, and energy conservation) is $\Delta\tau$ and $\Delta t \ll \Delta\tau$. The input to the DeepONet is the initial condition ($\boldsymbol{Y}_0$) of the state variables, and it predicts the dynamics of the state variables, $\boldsymbol{Y}(t)$ from $t = t_0 = 0$ to $t = t_{n_t} = n_t \Delta t$. The predicted state variable at $t = \Delta\tau = n_t \Delta t$ can be considered as input to the associated solver for the other physical system. We would like to mention that in the present study, we have not considered the integration with the CFD solver. However, we have discussed it as an example for illustrative purposes of our approach. Using the CFD application as an example, after the transport of the fluid is calculated by the CFD solver, the instantaneous states are considered as the initial conditions for the chemical kinetics. Typically, this would be an implicit numerical solver, possibly with additional heuristic modeling for partial combustion, but in this case becomes input to the DeepONet in lieu of the conventional numerical solution. Since in the present study, we did not consider the integration with CFD solver, we consider time decomposition of the system of ODEs given by Eq. (1) where the initial conditions of each segment are assumed to be known (generally input from the other associated solver). In Fig. 2, bottom, we have shown the time decomposition of the ODEs where, after every $t_{n_t} = n_t \Delta t$, the initial condition is known for each segment. Furthermore, in some cases, it is possible that the dynamics predicted by the DeepONet are not sufficiently advanced in time to be given as input to the other associated solver. In that case, we propose to consider a recursive prediction methodology in the neural operator where the output of the DeepONet at the last time instant of a segment is considered again as input to the DeepONet for the next segment. This process is continued until we have obtained sufficient progress in time for the chemical kinetics to be considered as input for the associated solver.

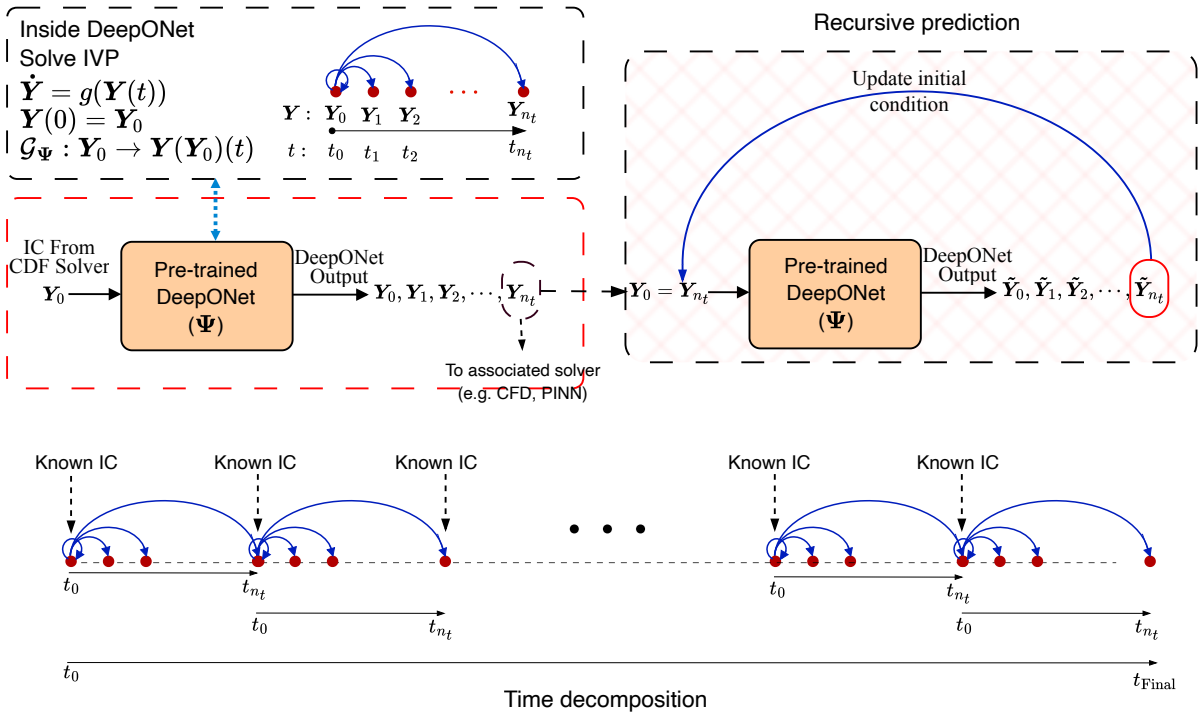


Fig. 2: **DeepONet for stiff chemical kinetics:** Schematic of the DeepONet prediction scheme considered in the present study. In the left middle, within the red dashed box, we have shown the prediction of the state variable using the pre-trained DeepONet parametrized with $\boldsymbol{\Psi}$. The input to the DeepONet is the initial condition of the state variables, $\boldsymbol{Y}_0$. The output is the predicted dynamics of the state variables, $\boldsymbol{Y}_0, \boldsymbol{Y}_1, \boldsymbol{Y}_2, \cdots \boldsymbol{Y}_{n_t}$. At the top right, we have shown the DeepONet learning process. It solves the initial value problem of the stiff system (Eq. (1) by mapping the initial condition to the dynamics of the state variables. Generally, the input for the DeepONet, the initial conditions, comes from the other associated solver, and the output of the DeepONet at a particular time instant goes to the other associated solver as input. Since we have not considered the DeepONet-CFD/PINN integration in the present study, we consider the time decomposition of the dynamic equation as shown at the bottom of the figure. In the time decomposition conditions, it is assumed that the initial conditions of the ODEs are known after every $t = t_{n_t} = n_t \Delta t$, which ideally comes from the associated solver. Thus, we will design our DeepONet to predict the dynamics considering time decomposition, assuming the initial conditions are known at every $t = t_{n_t} = n_t \Delta t$. Furthermore, when the DeepONet output is not sufficiently progressed in time to be input to the associated solver, we propose to consider a recursive prediction strategy as shown on the right within the black dashed box with the hatch. The output of the DeepONet at time instant $t = t_{n_t} = n_t \Delta t$ is considered as the input to the pretrained DeepONet, and the process continues till sufficient progress in time is achieved for the chemical kinetics to be input to the associated solver.

To obtain the objectives discussed above, we propose neural operator, particularly DeepONet [13] architectures which is based on the universal approximation theorem for operators [18]. In the next section Section 3, we will discuss in detail the proposed DeepONet architectures and the variations considered to obtain satisfactory prediction accuracy. We test our proposed operators with two computational examples that represent common chemical-kinetic reaction mechanisms. The first one is for the Syngas (Synthetic gas) mechanism, and the second is the GRI-Mech 3.0 mechanism. More details for the data generation for each of these cases are given in Section 4 and Section 5.

3 Methodology

We will use the AMORE framework introduced in the previous section to meet our objective for accurately predicting the stiff chemical kinetics described by a system of homogenous ODEs, i.e., Eq. (1). We will consider DeepONet as the neural operator and propose two adaptive loss functions. With n species involved in the system, the total number of state variables is $j = n+1$. These state variables consist of the temperature of the system and the mass fractions of n species. We will design our neural operator to predict the dynamics of $j = n+1$ state variables from the initial condition of these $j = n+1$ state variables. In the subsequent sections, we shall discuss the operators considered in this study, i.e., the DeepONet and its variants, including different loss functions formulated to enhance the accuracy of the operator learning. We will propose two adaptive loss functions. We will also discuss two training paradigms of DeepONet: the standard training, where both trunk and branch are trained together, and the two-step training, where the trunk and branch networks are trained separately in a sequential manner. We will also propose a mass-conserving operator that automatically satisfies the conservation of mass. AMORE is a general framework and can be used with other neural operators as well. We will also implement AMORE with Fourier Neural Operator (FNO) [31] as a case study. In Table 1, we have tabulated various methods considered in this study and their respective sections where these methods are discussed in detail.

Table 1: Different methods considered in the present study and the corresponding sections where the methods are discussed in detail.

	Description	Section
DeepONet (Single-step training)	Deep Operator Network including DeepONet and DeepOKAN	§3.1
	Partition of Unity (PoU) and Conservation of mass (CoM)	§3.2
	Adaptive loss functions	§3.3
Two-step DeepONet	Two-step training of DeepONet/DeepOKAN along with adaptive loss functions for trunk and branch training	§3.4
	Mass conserving neural operator	§3.5
	Recursive/Autoregressive prediction	§3.6
FNO	AMORE implementation in FNO	§6
ResNet	Discussion on ResNet model considered in the trunk and branch network	§A.1
KAN	Discussion on KAN model considered in the trunk network	§A.2

3.1 Deep Operator Network (DeepONet)

To learn the dynamics of the chemical kinetics governed by the system of stiff ODEs given by Eq. (1), we designed a DeepONet [13] with a single trunk network and a single branch network. The DeepONet predicts the dynamics of the multiple state variables of the stiff ODEs, i.e., the temperature and mass fractions of the species involved in the reaction. Thus, the output of the DeepONet is $\boldsymbol{Y}(t) = [y_0(t), y_1(t), y_2(t), \dots y_n(t)]^\top$, where y_0 is the temperature (T), and $y_1, y_2, \dots, y_n$ are the mass fractions of the n number of species participating in the chemical process. The branch takes the initial conditions, i.e., initial temperature, and the initial mass fractions of the species as input. Thus, the branch input is $\boldsymbol{Y}_0 = [y_0, y_1, y_2, \dots y_n]_0^\top$, while the trunk takes the time t as the input where the outputs need to be predicted. Thus, the operator learning can be written as

$$\boldsymbol{Y}(t) \approx \mathcal{G}_{\boldsymbol{\Psi}} : \boldsymbol{Y}_0 \mapsto \boldsymbol{Y}(\boldsymbol{Y}_0)(t), \tag{3}$$

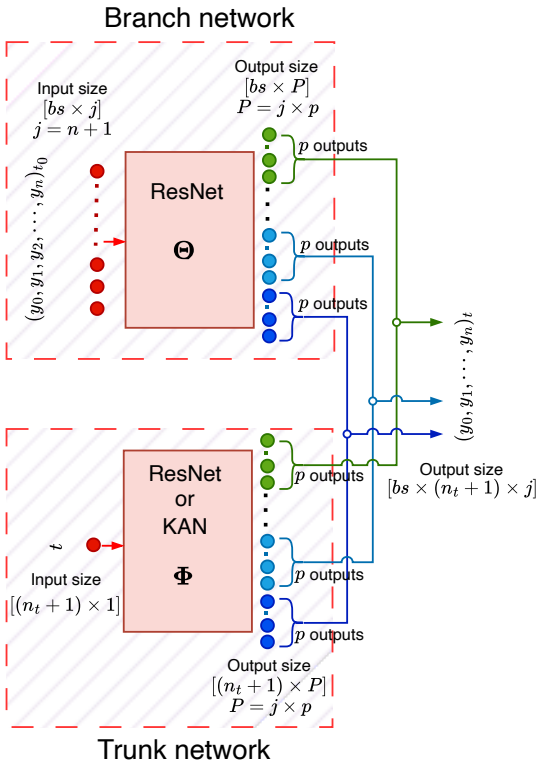
where $\boldsymbol{\Psi}$ are the parameters of the DeepONet. A dot product between the output of the branch and the trunk networks gives the output of the DeepONet. As discussed earlier, we will refer to the temperature and species' mass fractions together as state variables. Since we are predicting $j = n+1$ state variables from a single DeepONet, we divide the output neurons of the branch and trunk networks into $j = n+1$ equal parts, each part corresponding to one state variable. Thus, the approximation for the dynamics of the state variables using DeepONet can be written as

$$y_m(t) \approx \mathcal{G}_{\boldsymbol{\Theta},\boldsymbol{\Phi}} = \sum_{i=m\times p+1}^{(m+1)\times p} br_i(\boldsymbol{\Theta}) tr(\boldsymbol{\Phi})_i, \quad m = 0, 1, 2, \dots, n \tag{4}$$

where $br(\boldsymbol{\Theta}) \in \mathbb{R}^{bs\times(n+1)p}$ and $tr(\boldsymbol{\Phi}) \in \mathbb{R}^{(n_t+1)\times(n+1)p}$ are the output of the branch and trunk with $P = (n+1)p = jp$ output neurons, i.e., we consider p individual basis functions and coefficients for each state variable. $\boldsymbol{\Theta}$ and $\boldsymbol{\Phi}$ are the parameters of the branch and trunk network, respectively, i.e., $\boldsymbol{\Psi} = \{\boldsymbol{\Theta}, \boldsymbol{\Phi}\}$. We can write Eq. (4) in tensor notation by reshaping the output of branch $br(\boldsymbol{\Theta})$ to $\boldsymbol{B}$ and trunk $tr(\boldsymbol{\Phi})$ to $\boldsymbol{C}$ and using Einstein summation as

$$\begin{aligned} \boldsymbol{Y} &\approx \mathcal{G}_{\boldsymbol{\Theta},\boldsymbol{\Phi}} = \text{einsum}(ijk, ljk \rightarrow ilj, \boldsymbol{B}, \boldsymbol{C}), \\ &\boldsymbol{Y} \in \mathbb{R}^{bs\times(n_t+1)\times(n+1)}, \quad \boldsymbol{B} \in \mathbb{R}^{bs\times(n+1)\times p}, \quad \boldsymbol{C} \in \mathbb{R}^{(n_t+1)\times(n+1)\times p} \end{aligned} \tag{5}$$

where in the Einstein summation, the i indicates the number of batches in the branch (sample size), j indicates the number of state variables, k indicates the number of basis functions for each state variable, and l indicates the number of time points where the output has to be evaluated (trunk input). Furthermore, $j \times k$ is the number of output neurons in the branch and trunk networks. For convenience, we have not indicated the parameters $\boldsymbol{\Theta}$ and $\boldsymbol{\Phi}$ in $\boldsymbol{B}$ and $\boldsymbol{C}$. Furthermore, $\boldsymbol{Y}$ is a 3D tensor of size $bs \times (n_t+1) \times (n+1)$, and we consider that the batch (samples) vary from $bs \sim 1, 2, \cdots bs$, the time varies from $n_1 + 1 \sim 0, \Delta t, 2\Delta t, \cdots, n_t \Delta t$, and the state variables from $n+1 \sim 0, 1, 2, \cdots j-1$ (i.e., from $n+1 \sim 0, 1, 2, \cdots n$). As discussed earlier, the DeepONet architecture does not restrict the type of neural network architecture considered in the branch and trunk networks. In the present study, we consider two types of DeepONet architectures. In the first case, we consider ResNet (with MLP) in both the branch and trunk networks, which we refer to as DeepONet, in short, DON. In the second case, we consider ResNet in the branch while we consider KAN in the trunk network, which we call DeepOKAN, in short, DOK. We will use the term DeepONet as a general term to indicate both DON and DOK. A ResNet consists of residual blocks where there is a skipped connection after every two layers. The basic idea of KAN is that a multivariate continuous function can be represented as a sum of univariate functions [27]. A brief discussion on ResNet and KAN architecture is included in Appendix A. A schematic representation of the proposed DeepONet architecture is shown in Fig. 3 with input and output data structures.



Output

$$y_m(t) \approx \mathcal{G}_{\boldsymbol{\Theta},\boldsymbol{\Phi}} = \sum_{i=m\times p+1}^{(m+1)\times p} br_i(\boldsymbol{\Theta}) tr_i(\boldsymbol{\Phi}),\ m = 0, 1, 2, \ldots, n$$

- $br(\boldsymbol{\Theta})$: Branch output
- $tr(\boldsymbol{\Phi})$: Trunk output
- p : Number of basis function for each state variable
- n : Number of species
- $j = n + 1$: Number of state variable (Temperature and species mass fraction)

In tensor form

Reshape output of branch and trunk

- $br(\boldsymbol{\Theta}) \rightarrow \boldsymbol{B}$
  - Shape of $br(\boldsymbol{\Theta})$: $bs \times (j \times p)$
  - Shape of $\boldsymbol{B}$: $bs \times j \times p$
- $tr(\boldsymbol{\Phi}) \rightarrow \boldsymbol{C}$
  - Shape of $tr(\boldsymbol{\Phi})$: $(n_t + 1) \times (j \times p)$
  - Shape of $\boldsymbol{C}$: $(n_t + 1) \times j \times p$

Use Einstein summation

- $\boldsymbol{Y} = \text{einsum}(ijk, ljk \rightarrow ilj, \boldsymbol{B}, \boldsymbol{C})$
- Shape of $\boldsymbol{Y}$: $bs \times (n_t + 1) \times j$

Optimization problem

$$\boldsymbol{\Theta}^*, \boldsymbol{\Phi}^* = \min_{\boldsymbol{\Theta},\boldsymbol{\Phi}} \mathcal{L}(\boldsymbol{\Theta}, \boldsymbol{\Phi}) = \min_{\boldsymbol{\Theta},\boldsymbol{\Phi}} \|\boldsymbol{Y}_{\text{data}} - \boldsymbol{Y}(\boldsymbol{\Theta}, \boldsymbol{\Phi})\|_k$$

Fig. 3: **Schematic diagram of the proposed DeepONet** architecture showing branch and trunk networks and multiple outputs. We have shown the branch network in the top left in the red dashed box. The inputs to the branch are the initial conditions of the state variables $\boldsymbol{Y}_0 \in \mathbb{R}^{bs\times j}$ and the outputs of the branch are $br(\boldsymbol{\Theta}) \in \mathbb{R}^{bs\times P}$. Where bs and j are the number of samples and the number of state variables, respectively, and P is the number of output neurons. The input to the trunk is time $t \in \mathbb{R}^{(n_t+1)\times 1}$ and the outputs of the trunk are $tr(\boldsymbol{\Phi}) \in \mathbb{R}^{(n_t+1)\times P}$. Where $(n_t + 1)$ is the number of time points where the output needs to be predicted. In the present study, we have considered ResNet (with trainable parameters $\boldsymbol{\Theta}$) as a choice of network architecture for the branch network and ResNet or KAN (with trainable parameters $\boldsymbol{\Phi}$) as a choice of network architecture for the trunk network. A brief discussion on the ResNet and KAN architectures is included in Appendix A. The branch and trunk network each has $P = j \times p$ neurons in the output layers, where j is the number of state variables and p is the number of basis functions and coefficients considered to approximate each variable. The output $\boldsymbol{Y} \in \mathbb{R}^{bs\times(n_t+1)\times j}$ is given by the dot product between the output of the branch and trunk networks, and is shown on the right side in standard form and tensor form. We also implemented the partition of unity in the trunk network, and the implementation is discussed in Section 3.2.1. Furthermore, we also consider a loss due to the conservation of mass along with data loss, which is discussed in Section 3.2.2. As discussed in the AMORE framework, we propose two adaptive loss functions, and these are discussed in Section 3.3.

The optimal parameters of the DeepONet ($\boldsymbol{\Psi} = \{\boldsymbol{\Theta}, \boldsymbol{\Phi}\}$) can be obtained by minimizing a loss function,

$$\boldsymbol{\Theta}^*, \boldsymbol{\Phi}^* = \min_{\boldsymbol{\Theta},\boldsymbol{\Phi}} \mathcal{L}(\boldsymbol{\Theta}, \boldsymbol{\Phi}) = \min_{\boldsymbol{\Theta},\boldsymbol{\Phi}} \|\boldsymbol{Y}_{\text{data}} - \boldsymbol{Y}(\boldsymbol{\Theta}, \boldsymbol{\Phi})\|_k, \tag{6}$$

where $\boldsymbol{Y}_{\text{data}}$ is the labeled data and the $\boldsymbol{Y}(\boldsymbol{\Theta}, \boldsymbol{\Phi})$ is the DeepONet prediction, $\|\cdot\|_k$ is a chosen norm to measure the discrepancy between the two. For convenience, from this point forward, we will denote the true value as $\boldsymbol{Y}$ and the predicted value as $\widehat{\boldsymbol{Y}}$ (with a hat). In the present study, for the optimization process, we consider the Adam [32] optimizer in a Tensorflow version 2 environment with 64-bit data precision. We consider the data loss as the mean square of discrepancies between the predicted and true values,

$$\mathcal{L}_{\text{data}}(\boldsymbol{\Theta}, \boldsymbol{\Phi}) = \frac{1}{j \times bs \times (n_t + 1)} \sum_{a=0}^{j-1} \sum_{b=1}^{bs} \sum_{c=0}^{n_t} \left(Y_{bca} - \widehat{Y}_{bca}\right)^2, \tag{7}$$

$j \rightarrow$ No. of states,
$n_t + 1 \rightarrow$ No. of time points
$bs \rightarrow$ No. of samples

We also like to mention that we consider the subscript for time point and species starts from zero (0) to match the previous nomenclature used. In the subsequent section, we will discuss the loss function again and define a composite (with conservation of mass) loss function and two adaptive loss functions (Section 3.3). Furthermore, we also consider the Partition of Unity (PoU) in the trunk network. A detailed discussion on the implementation is given in Section 3.2.1. To assess the accuracy and efficiency of the operators, we will consider the relative L_2 errors, which we discuss in more detail in Section 3.7.

3.2 Partition of Unity and conservation of mass

3.2.1 Partition of Unity (PoU)

Partition of unity (PoU) [33] allows us to stitch locally well-defined approximation spaces to give them a conforming global definition of desired regularity. PoU is defined such that the basis functions are bounded $[0, 1]$ and the sum of the basis functions at any point is one, i.e., for the trunk output for each state variable,

$$0 \leq tr_i(t) \leq 1, \quad \forall\, t \in [0, n_t \Delta t], \quad \forall\, i \in [1, p] \tag{8a}$$

$$\sum_{i}^{p} tr_i(t) = 1, \quad \forall\, t \in [0, n_t \Delta t] \tag{8b}$$

Lee et al. [34], Trask et al. [35] consider PoU in a neural network by considering a softmax function in the last hidden layer of a neural network. Goswami et al. [14] considered PoU in DeepONet in the trunk network by incorporating a PoU-based additional loss function to satisfy the second equation of PoU, i.e., Eq. (8b) in the loss function as an additional penalty term. However, this does not ensure the bound ($[0, 1]$) of the basis function. Furthermore, this also does not ensure the summation of the basis functions to be one, as it depends on the penalty term considered in the loss function for the PoU-based loss. Sharma and Shankar [36] proposed an ensemble DeepONet wherein individual trunk networks are stacked column-wise to form an ensemble trunk along with a single branch network, as well as a PoU-mixture-of-experts (PoU-MoE) trunk architecture. The idea of PoU-MoE trunk is to partition the domain into overlapping spherical patches and employing a separate trunk on each of them, and then blending them into a single trunk network. The authors also noted that the PoU-MoE trunk incurs a higher computational cost due to the forward pass consisting of serial looping over the patches of the domain.

In the present study, we design our trunk network to automatically satisfy the PoU in the trunk network for each state variable, which are the basis functions in the approximation of the output. We consider the softmax function to enforce PoU in the trunk network. The output of the trunk network is passed through a softmax function, which ensures that both the basis functions are bounded in $(0, 1)$, ($0 < tr_i(t) < 1, \forall\, t \in [0, n_t \Delta t], \forall\, i \in [1, p]$) and the sum of the basis functions as one ($\sum_i^p tr_i(t) = 1, \ \forall\, t \in [0, n_t \Delta t]$). Since we have divided the output of the trunk into as many numbers as the number of state variables, we have enforced the PoU for each state variable. This is implemented by considering the softmax function to $\boldsymbol{C}$ before Eq. (5) along the basis dimension. Thus,

$$\boldsymbol{C}_{ljk} = \text{softmax}(\boldsymbol{C}_{ljk}, \ \text{axis} = k). \tag{9}$$

where l is the number of time points where the output has to be evaluated (trunk input), j is the number of state variables, and k is the number of basis functions for each state variable.

3.2.2 Conservation of mass

Under the assumption of a zero-dimensional reactor made in Section 2, we know that the chemical reactions are balanced, i.e., the total mass of the reacting agents and products at any time is the same, and the sum of mass fractions is always unity. Thus, we consider an additional loss function corresponding to the conservation of mass (CoM) defined as,

$$\mathcal{L}_{\text{CoM}}(\boldsymbol{\Theta}, \boldsymbol{\Phi}) = \frac{1}{bs \times (n_t + 1)} \sum_{b=1}^{bs} \sum_{c=0}^{n_t} \left(\sum_{a=1}^{j-1} \widehat{Y}_{bca} - 1.0 \right)^2, \tag{10}$$

$j \rightarrow$ No. of states variable,
$n_t + 1 \rightarrow$ No. of time points
$bs \rightarrow$ No. of samples

We would like to draw attention to the fact that the summation over the state variables starts from $a = 1$, as the state variable corresponding to $a = 0$ is temperature. Secondly, if any normalization scheme is considered, then the summation in Eq. (10) needs to be calculated in the physical domain. The total loss function is

$$\mathcal{L}(\boldsymbol{\Theta}, \boldsymbol{\Phi}) = \mathcal{L}_{\text{data}}(\boldsymbol{\Theta}, \boldsymbol{\Phi}) + \mathcal{W}_{\text{CoM}} \mathcal{L}_{\text{CoM}}(\boldsymbol{\Theta}, \boldsymbol{\Phi}) \tag{11}$$

where, $\mathcal{L}_{\text{data}}(\boldsymbol{\Theta}, \boldsymbol{\Phi})$ is the data loss discussed in Eq. (7) or later for adaptive data loss functions, and $\mathcal{L}_{\text{CoM}}(\boldsymbol{\Theta}, \boldsymbol{\Phi})$ is the loss due to conservation of mass and defined in Eq. (10), $\mathcal{W}_{\text{CoM}}$ is a relative weight term which regulates the contribution of the conservation of mass loss in the loss function ($\mathcal{L}(\boldsymbol{\Theta}, \boldsymbol{\Phi})$). We will discuss the weight, $\mathcal{W}_{\text{CoM}}$, in detail in the next section.

We would also like to highlight that we have considered the PoU and conservation of mass constraint only in the conventional (one-step) training of DeepONet. These are, however, not considered in the two-step training (Section 3.4).

3.3 Adaptive loss function

Most deep learning algorithms involve the minimization of a cost or loss function, characterized by the task to which they are employed. In many cases, it is observed that the error is not uniform over different data points. For instance, solving a forward problem using a physics-informed network (PINN) [9] involves residual, initial/boundary condition losses. The errors at different collocation points are different. To address this issue McClenny and Braga-Neto [37] proposed self-adaptive weights (SAW) in PINNs for each collocation, initial/boundary points. The idea of the self-adaptive weights is that the points where the loss is greater should be weighted more. The weights

are updated using a min-max problem with a gradient-based optimizer. While the loss is minimized with respect to network parameters, it is maximized with respect to the self-adaptive weights. Anagnostopoulos et al. [38] proposed residual-based attention (RBA), where instead of updating the adaptive weights using a gradient-based optimizer, the authors propose a gradient-free update scheme which is based on the residual from the previous epoch. The idea of self-adaptive weights in the loss functions has been extended to the DeepONet [39, 40] to enhance the robustness of these models. During training, while the gradient descent is performed on the network parameters in the loss function, the self-adaptive weights are updated by performing gradient ascent. In [39], each adaptive weight is associated with the corresponding evaluation point. Furthermore, in both cases (SAW, RBA), additional hyperparameters are needed, such as learning rate and decay parameters.

In the present study, the proposed DeepONet architecture predicts multiple state variables; consequently, the errors for each state variable often differ, sometimes in the order of magnitude. Thus, considering a single global data loss function defined in the previous section does not do justice to all the state variables. Therefore, to improve our DeepONet's accuracy, we propose two gradient-free adaptive weighting schemes in the loss function. The basic idea is that the state variables and the samples with the higher error values need to be penalized more in the loss function than the cases with the lower error values. We introduce the two adaptive loss functions with the aforementioned adaptation in Sections 3.3.1 and 3.3.2, and are called Type-A and Type-B, respectively. The proposed adaptive loss functions do not require any additional hyperparameters.

3.3.1 Adaptive loss function: Type-A

The first adaptive loss function we propose is based on the idea that the state variables with higher error values should have more weight in the loss function, so that they are penalized more heavily. Thus, we define the data loss function as Eq. (12), with adaptive weights $\mathcal{W}_a$,

$$\mathcal{L}_{\text{data}}(\boldsymbol{\Theta},\boldsymbol{\Phi}) = \sum_{a=0}^{j-1} \mathcal{W}_a \left[\frac{1}{bs \times (n_t+1)} \sum_{b=1}^{bs} \sum_{c=0}^{n_t} \left(Y_{bca} - \widehat{Y}_{bca} \right)^2 \right], \tag{12}$$

$j \rightarrow$ No. of states variable,
$n_t + 1 \rightarrow$ No. of time points
$bs \rightarrow$ No. of samples

such that $\sum_{a=0}^{j-1} \mathcal{W}_a = \mathcal{R}_1$, and $\mathcal{W}_a \geq 0$ for every epoch. The subscript in $\mathcal{W}_a$ are the weights associated with the state variable 'a', where $a = 0, 1, \ldots, j-1$. We proposed a gradient-free updating scheme with no additional hyper-parameter requirement to update the weights. The weights are updated based on the relative error between the state variables in the previous epoch, such that

$$\mathcal{W}_a = \frac{X_a}{\sum_{d=0}^{j-1} X_d} \times \mathcal{R}_1 \tag{13}$$

We consider X_a (and X_d) as the mean (calculated over samples) of the relative L_2 error (calculated over time points) in the physical space for each state variable, i.e., we calculate the relative L_2 error in the time direction $n_t + 1$ and then take the mean in the batch direction bs for each state variable. This way, we calculate j means of the relative L_2 error corresponding to j state variables. We initialize the weights with an equal value of one, i.e., $\mathcal{W}_a = 1$. Thus, $\sum_{a=0}^{j-1} \mathcal{W}_a = \mathcal{R}_1 = j$, which is the number of state variables. We also note that the limiting value of $\mathcal{W}_a$ can be zero only when X_a is zero, i.e., the relative L_2 error is zero. We also assume that all X_a are not zero at the same epoch so that the denominator of Eq. (13) is non-zero. Furthermore, the sum of all the weights $\mathcal{W}_a$ at any epoch is always $\mathcal{R}_1$ as shown in Eq. (B.5) (in Appendix B.2).

3.3.2 Adaptive loss function: Type-B

We propose a second variant of the adaptive loss function based on the idea that, along with the state variables, the samples with higher error values should be given more weight, hence penalized more heavily, in the loss function. Thus, we defined the data loss function as Eq. (14), with an adaptive weight $\mathcal{W}_{ba}$,

$$\mathcal{L}_{\text{data}}(\boldsymbol{\Theta},\boldsymbol{\Phi}) = \sum_{a=0}^{j-1} \sum_{b=1}^{bs} \mathcal{W}_{ba} \left[\frac{1}{n_t+1} \sum_{c=0}^{n_t} \left(Y_{bca} - \widehat{Y}_{bca} \right)^2 \right], \tag{14}$$

$j \rightarrow$ No. of states variable,
$n_t + 1 \rightarrow$ No. of time points
$bs \rightarrow$ No. of samples

such that $\sum_{a=0}^{j-1} \sum_{b=1}^{bs} \mathcal{W}_{ba} = \mathcal{R}_2$, and $\mathcal{W}_{ba} \geq 0$ for every epoch. $\mathcal{W}_{ba}$ represents the weights associated with the state variable a and the sample b. Similar to "Type-A" adaptive loss, the weights $\mathcal{W}_{ba}$ are updated using a gradient-free algorithm. The weights are updated based on the relative error between the state variables and samples in the previous epoch, such that

$$\mathcal{W}_{ba} = \frac{X_{ba}}{\sum_{d=0}^{j-1} \sum_{e=1}^{bs} X_{ed}} \times \mathcal{R}_2 \tag{15}$$

We consider X_{ba} (and X_{ed}) as the relative L_2 error (calculated over time-points) in the physical space for each state variable and sample. In this way, we calculate $j \times bs$ relative L_2 error corresponding to the j state variables

and bs samples. Similar to "Type-A", we initialize the weights with an equal value of one, i.e., $\mathcal{W}_{ba} = 1$. Thus, $\sum_{a=0}^{j-1}\sum_{b=1}^{bs}\mathcal{W}_{ba} = \mathcal{R}_2 = j \times bs$. The limiting value of $\mathcal{W}_{ba}$ can be zero only when $X_{ba} = 0$. We assume that all X_{ba} are not zero so that the denominator of Eq. (14) is non-zero. Similar to Eq. (B.5), we can show that the sum of all $\mathcal{W}_{ba}$ at any epoch is always $\mathcal{R}_2$.

A brief discussion on the calculation of X_a and X_{ba} is given in Appendix B.2. As discussed earlier, we also consider the two-step training of DeepONet. Appropriate modifications are needed in the adaptive loss functions to incorporate them in two-step training. These modifications are discussed in the implementation of two-step training of DeepONet in Section 3.4. Furthermore, as discussed in Section 3.2.2, the weight $\mathcal{W}_{\text{CoM}}$ acts as a weighting coefficient for the loss due to the conservation of mass ($\mathcal{L}_{\text{CoM}}(\boldsymbol{\Theta}, \boldsymbol{\Phi})$). In the cases where both data loss and loss due to mass conservation are considered, we consider $\mathcal{W}_{\text{CoM}}$ as a multiplier of the weights ($\mathcal{W}_a$ or $\mathcal{W}_{ab}$) of the data loss.

3.4 Two-step training of DeepONet

The training method for DeepONet discussed so far considers training the parameters of the branch ($\boldsymbol{\Theta}$) and trunk ($\boldsymbol{\Phi}$) together as shown in Eq. (6). Lee and Shin [30] proposed a novel reparameterization and training strategy for DeepONet. The method consists of reparameterizing the existing DeepONet architecture and a two-step training of DeepONet, where the parameters of the trunk and branch are trained in a sequential manner rather than together. The parameters of the trunk are trained first, followed by training the parameters of the branch. Lee and Shin [30] considered DeepONet with a single output variable. Peyvan et al. [41] considered DeepONet with multiple output variables. The authors considered the same basis function for all the output variables with different coefficients from a single-branch network. Kiyani et al. [23] also considered multiple output variables with the same basis function in a trunk in crack propagation problems. The authors considered independent branches, one for each output variable. Jin et al. [42] considered two-step training of DeepONet for multiple outputs with a single trunk with different basis functions. The coefficients of the basis functions are obtained from a composite branch with an independent MLP for each output variable.

In the present study, we consider a generalized form of two-step training of DeepONet for multiple output variables, with one trunk but different basis functions for each output variable and one branch but different coefficients for each output variable. This architecture, with different basis functions for each output variable from a single trunk and different coefficients for each output variable from a single branch, is important for the scalability of the method. Furthermore, to improve the accuracy of DeepONet using two-step training, we extend the adaptive loss functions discussed in the previous section to trunk and branch training with appropriate modifications. It is important to note that the loss due to the conservation of mass has not been considered in the two-step training of DeepONet. This is owing to the fact that within the two-step training framework, while the trunk is trained using labeled dataset to learn the basis representation of the output along with its respective coefficients, the branch is trained using the reconstructed output obtained from trunk training. Moreover, in this case, we have not considered the PoU in the trunk network either as QR decomposition of the trunk output is performed in the two-step trained DeepONet.

3.4.1 Training of trunk network

The first step in two-step training is the training of the parameters of the trunk ($\boldsymbol{\Phi}$) and a trainable tensor $\boldsymbol{A}$ through a minimization problem

$$\boldsymbol{\Phi}^*, \boldsymbol{A}^* = \min_{\boldsymbol{\Phi}, \boldsymbol{A}} \mathcal{L}(\boldsymbol{\Phi}, \boldsymbol{A}) = \min_{\boldsymbol{\Phi}, \boldsymbol{A}} \|\boldsymbol{Y}_{\text{data}} - \boldsymbol{Y}(\boldsymbol{\Phi}, \boldsymbol{A})\|_k, \tag{16}$$

where $\boldsymbol{Y}_{\text{data}}$ is the labeled (DeepONet output training) data and $\boldsymbol{Y}(\boldsymbol{\Phi}, \boldsymbol{A})$ is the predicted output from the trunk multiplied by a trainable tensor $\boldsymbol{A}$. Since we predict multiple outputs from a single DeepONet, let us discuss this multiplication before proceeding. We have shown a schematic diagram for the two-step training of DeepONet in Fig. B.1 (in Appendix B.1). For the trunk training, each of the output state variables is approximated as

$$\begin{aligned} y_m(t) &\approx tr(\boldsymbol{\Phi})\,[:, m \times p + 1 : (m+1) \times p]\,\boldsymbol{A}_m, \qquad m = 0, 1, 2, \ldots n, \\ & tr(\boldsymbol{\Phi})[:, m \times p + 1 : (m+1) \times p] \in \mathbb{R}^{(n_t+1)\times p}, \quad \boldsymbol{A}_m \in \mathbb{R}^{p \times bs} \end{aligned} \tag{17}$$

where $tr(\boldsymbol{\Phi}) \in \mathbb{R}^{(n_t+1)\times(n+1)p}$ is the output of the trunk with $P = (n+1)p = jp$ output neurons, i.e., we consider p individual basis functions for each state variable, $\boldsymbol{A} \in \mathbb{R}^{(n+1)\times p \times bs}$ is a trainable tensor, where bs is the number of samples and j is the number of state variables.

Similar to the formulation discussed in Section 3.1 for one step (conventional) training of DeepONet, we can rewrite the Eq. (17) in tensor form using the Einstein summation. We reshape the output of the trunk network $tr(\boldsymbol{\Phi})$ to $\boldsymbol{C}$ and using Einstein summation as

$$\begin{aligned} \boldsymbol{Y} \approx \boldsymbol{Y}(\boldsymbol{\Phi}, \boldsymbol{A}) &= \text{einsum}(ijk, jkl \rightarrow lij, \boldsymbol{C}, \boldsymbol{A}), \\ & \boldsymbol{Y} \in \mathbb{R}^{bs\times(n_t+1)\times(n+1)}, \quad \boldsymbol{C} \in \mathbb{R}^{(n_t+1)\times(n+1)\times p}, \quad \boldsymbol{A} \in \mathbb{R}^{(n+1)\times p\times bs} \end{aligned} \tag{18}$$

where in the Einstein summation, the i is the number of time points (trunk input), j is the number of state variables, k is the number of basis functions for each state variable, and l is the number of samples (i.e., number of time histories per batch)

We optimize the parameters of the trunk ($\boldsymbol{\Phi}$) and $\boldsymbol{A}$ using an appropriate loss function as shown in Eq. (16). We consider three different loss functions, one non-adaptive and two adaptive loss functions similar to those discussed in one-step training given by Eqs. (7), (12) and (14) where the Y_{bca} is the labeled output data corresponding to different samples and $\widehat{Y}_{bca}$ is the prediction given by Eq. (18). The weights in the adaptive loss function case were calculated as discussed in Sections 3.3.1 and 3.3.2 for adaptive loss Type-A and Type-B, respectively. We used the Adam [32] optimizer in a Tensorflow version 2 environment with 64-bit data precision to optimize the trunk parameters $\boldsymbol{\Phi}$ and $\boldsymbol{A}$. We also assume $p < n_t$ and $tr(\boldsymbol{\Phi}^*)$ is full-rank for each output variable [30].

3.4.2 Training of branch network

After the training of the trunk, the next step is the training of the branch network. The parameters of the branch network ($\boldsymbol{\Theta}$) are optimized considering a data loss function. The labeled data for the loss function is obtained from a reconstructed output from the trunk output $\boldsymbol{U} = \boldsymbol{R}^*\boldsymbol{A}^*$, where $\boldsymbol{R}^*$ is obtained from the QR decomposition of the trunk output. We also consider $\boldsymbol{A}^*$ from the least-squares approximation of the trunk output and labeled data. As we are predicting multiple outputs, let us discuss the process for calculating the labeled data ($\boldsymbol{U} = \boldsymbol{R}^*\boldsymbol{A}^*$) for the branch training. The QR decomposition of the trunk for each state variable can be written as,

$$\boldsymbol{Q}^*_m\boldsymbol{R}^*_m = \mathrm{QR}\left(tr(\boldsymbol{\Phi}^*)[:,\, m\times p+1 : (m+1)\times p]\right), \qquad m = 0,1,2,\cdots,n \tag{19}$$

and the least-square approximation of the $\boldsymbol{A}^*$ can be written as

$$\boldsymbol{A}^*_m = \mathrm{LSM}\left(tr(\boldsymbol{\Phi}^*)[:,\, m\times p+1 : (m+1)\times p],\, y_m\right), \qquad m = 0,1,2,\cdots,n \tag{20}$$

where $y_m \in \mathbb{R}^{(n_t+1)\times bs}$ is the labeled data corresponding to m^{th} state variables. Note that we consider the $\boldsymbol{A}^*$ using Eq. (20) instead of the optimized $\boldsymbol{A}^*$ from Eq. (16), as in this case, we observed smaller error in the trunk error. The branch network is trained by minimizing a loss function

$$\mathcal{L}(\boldsymbol{\Theta}) = \|br(\boldsymbol{\Theta}) - \boldsymbol{U}\| \tag{21}$$

where $br(\boldsymbol{\Theta}) \in \mathbb{R}^{bs\times(n+1)p}$ and $\boldsymbol{U}$ is reconstructed data calculated from the output of the trunk network,

$$\boldsymbol{U}_m = \boldsymbol{R}^*_m\boldsymbol{A}^*_m, \qquad m = 0,1,2,\cdots,n \tag{22}$$

We rewrite the above in tensor notation by reshaping the branch output. The $\boldsymbol{U} \in \mathbb{R}^{(n+1)\times p\times bs}$, and $br(\boldsymbol{\Theta}) \in \mathbb{R}^{bs\times(n+1)p}$. We reshape the branch output $br(\boldsymbol{\Theta})$ to $\boldsymbol{B} \in \mathbb{R}^{bs\times(n+1)\times p}$, the axis of $\boldsymbol{U}$ is changed to $\boldsymbol{U} \in \mathbb{R}^{bs\times(n+1)\times p}$.

The parameters ($\boldsymbol{\Theta}$) of the branch are optimized using a loss function between the branch output and $\boldsymbol{U}$,

$$\boldsymbol{\Theta}^* = \min_{\boldsymbol{\Theta}} \mathcal{L}(\boldsymbol{\Theta}) = \min_{\boldsymbol{\Theta}} \|\boldsymbol{B}(\boldsymbol{\Theta}) - \boldsymbol{U}\|_k \tag{23}$$

Similar to previous optimization processes, we consider three different loss functions: the MSE loss function and two adaptive loss functions. However, in this case, marginal modifications are required in the adaptive loss functions. These are discussed in Appendix B.1.

3.4.3 Prediction using trained DeepONet

In the previous sections, we have discussed the training of the trunk and branch using the two-step training method of DeepONet, where the parameters of the trunk and branch networks are trained sequentially rather than together. The output of DeepONet, in this case, is approximated as [30],

$$\begin{aligned} y_m = tr(\boldsymbol{\Phi})[:, m\times p+1 : (m+1)\times p]\boldsymbol{T}_m br^\top(\boldsymbol{\Theta})[:, m\times p+1 : (m+1)\times p], \\ \text{where} \quad \boldsymbol{T}_m = \boldsymbol{R}^{*-1}_m, \;\; m = 0,1,2\cdots,n \end{aligned} \tag{24}$$

where $tr(\boldsymbol{\Phi})[:, m\times p+1 : (m+1)\times p] \in \mathbb{R}^{(n_t+1)\times p}$ and $br(\boldsymbol{\Theta})[:, m\times p+1 : (m+1)\times p] \in \mathbb{R}^{bs\times p}$. In the present study, we consider that the outputs are predicted at the same temporal points for all the samples during training and testing. Thus, we consider a simplified version of Eq. (24) as,

$$= \boldsymbol{Q}^*_m\boldsymbol{R}^*_m\boldsymbol{T}_m br^\top(\boldsymbol{\Theta})[:, m\times p+1 : (m+1)\times p]) \tag{25a}$$

$$= \boldsymbol{Q}^*_m br^\top(\boldsymbol{\Theta})[:, m\times p+1 : (m+1)\times p], \quad \text{since} \quad \boldsymbol{R}^*_m\boldsymbol{T}_m = \boldsymbol{I} \tag{25b}$$

where $\boldsymbol{Q}^*_m \in \mathbb{R}^{(n_t+1)\times p}$ and $\boldsymbol{R}^*_m \in \mathbb{R}^{p\times p}$. We can interpret Eq. (25b) as $\boldsymbol{Q}^*_m$ are the orthogonal basis functions obtained from the output dataset, and the branch gives the coefficient to these basis functions. We can rewrite Eq. (25b) in a tensor form by reshaping the output of the branch and using the Einstein summation,

$$\begin{aligned} \boldsymbol{Y} \approx \mathcal{G}_{\boldsymbol{\Theta},\boldsymbol{\Phi}} = \mathrm{einsum}(ijk, jlk \rightarrow ilj, \boldsymbol{B}, \boldsymbol{Q}^*), \\ \boldsymbol{Y} \in \mathbb{R}^{bs\times(n_t+1)\times(n+1)}, \quad \boldsymbol{B} \in \mathbb{R}^{bs\times(n+1)\times p}, \quad \boldsymbol{Q}^* \in \mathbb{R}^{(n+1)\times(n_t+1)\times p} \end{aligned} \tag{26}$$

where in the Einstein summation, the i is the number of batches in the branch (sample size), j is the number of state variables, k is the number of basis functions for each species, and l is the number of time points where the output has to be evaluated (trunk input). We would also like to mention that in the calculation of the weights $\mathcal{W}_a$ and $\mathcal{W}_{ba}$ in the case of branch training, we use the predicted $\boldsymbol{Y}$ using Eq. (26) for relative L_2 error calculation associated with each state variables and samples. In this case of weight calculation, the $\boldsymbol{Q}^*$ is constant (fixed), and the branch outputs $\boldsymbol{B}$ change with epochs.

3.5 Mass conserving DeepONet

In the previous sections, we have proposed DeepONet for stiff chemical systems, including two-step training and adaptive loss functions. In Section 3.2.2, we have proposed the conservation of mass for stiff chemical systems by incorporating an additional loss function due to the conservation of mass. However, the accuracy depends on the penalty imposed on the loss function, and the conservation of mass is enforced only weakly. Furthermore, we did not consider the conservation of mass in the two-step training of DeepONet. In this section, we will develop a DeepONet model that automatically satisfies the conservation of mass.

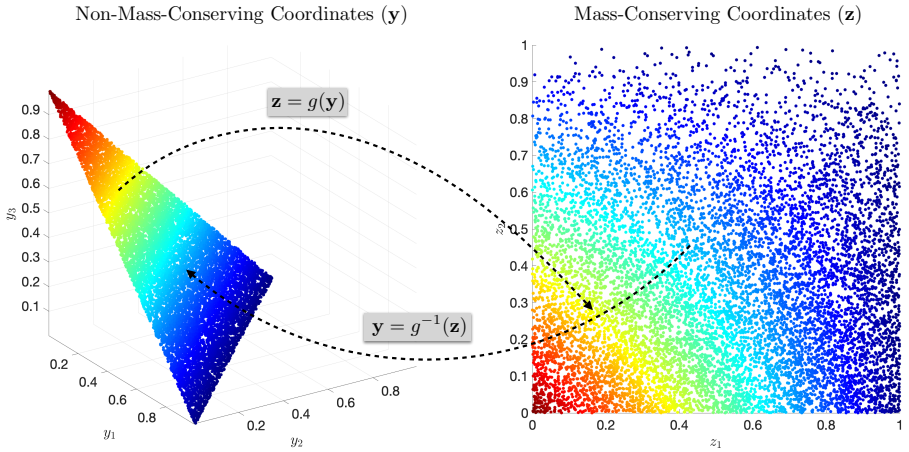


Fig. 4: **Mass conserving map** showing the species mass fraction vector ($\boldsymbol{y} \in \mathbb{R}^n$) is mapped uniquely to a mass-conserving coordinate ($\boldsymbol{z} \in \mathbb{R}^{n-1}$) and vice versa.

The conservation of mass is expressed as a linear constraint on the mass fraction

$$\sum_{i=1}^{n} y_i = 1. \tag{27}$$

where $0 \leq y_i \leq 1$. As a result of the above constraint, the mass fraction vector $\boldsymbol{y}$ cannot lie anywhere within the n-dimensional hypercube; instead, it must reside within a simplex to ensure the conservation of mass.

Given the above constraint, it is evident that the species mass fraction vector can be encoded, without any loss of information, with an $(n-1)$-dimensional vector $\mathbf{z} = (z_1, z_2, \cdots, z_{n-1})$ as follows:

$$z_1, z_2, \cdots z_{n-1} = g(y_1, y_2, \cdots y_n) \tag{28}$$

where the function $g(.)$ can be considered as an encoder of the input data. For reasons that will become clearer later in this section, we also require that the function $g(.)$ be invertible, such that

$$y_1, y_2, \cdots y_n = g^{-1}(z_1, z_2, \cdots z_{n-1}) \tag{29}$$

where $0 \leq z_i \leq 1$. Therefore, any mass fraction vector ($\boldsymbol{y}$) can be uniquely mapped to a unique vector ($\boldsymbol{z}$) and vice versa. The schematic of such mapping for $n = 3$ species is shown in Fig. 4. We construct the maps using the popular spectral element simulation of PDES [43] and discuss it in detail in Appendix B.3

3.6 Recursive prediction

As discussed in previous sections, the objective of the present study is to develop an accurate neural operator for stiff chemical kinetics so that it can be integrated with a CFD solver in future studies. In this context, it is possible that the time series predicted by the neural operator may not progress sufficiently in time to be considered as input to the CFD solver. In such a case, we propose considering a recursive scheme which predicts the output auto-regressively from the prediction.

As discussed in Section 2, Fig. 2, the pre-trained neural operator predicts the time series from $t = 0$ to $t = n_t \Delta t$. The output states at time $n_t \Delta t$, i.e., $\boldsymbol{Y}_{n_t}$ are considered as input to the same pre-trained neural operator again to predict the next time segment of the same length as the initial segment. The process continues until the predicted time series has progressed sufficiently to be considered as input to the CFD solver. We would like to mention two important aspects in this case. The first aspect is that an appropriate normalization and denormalization scheme must be considered while performing a recursive prediction. The second one being that the input to the trunk is the same $t = 0, \Delta t, 2\Delta t, \cdots, n_t \Delta t$ in all recursive predictions. We can consider it as translating the time axis such that the end point of the previous time series becomes the first point of the newly translated time series. In the present study, since we have not integrated our operator with the CFD solver, we perform recursive predictions for the entire time series.

3.7 Error metric

In order to check the accuracy of DeepONet during training and testing, we consider the relative L_2 error defined as,

$$\text{Relative } L_2 \text{ error} = \frac{||y - \hat{y}||_2}{||y||_2} = \frac{\sqrt{\sum_{i=0}^{n_t} (y_i - \hat{y}_i)^2}}{\sqrt{\sum_{i=0}^{n_t} y_i^2}}, \tag{30}$$

where $n_t + 1$ is the number of time points considered. It is important to note that the output of DeepONet and the true data are 3-dimensional, with size $bs \times (n_t + 1) \times j$, where bs, $n_t + 1$, and j are the sample size, number of time points, and number of state variables, respectively. In Eq. (30), we consider the relative L_2 error only in the direction of time points. Thus, to get a global error, we consider the mean of the relative L_2 error for each state variable and take the mean again to obtain an overall scalar error value, which is the global error. However, calculating a global error in this way may not reflect the robustness of the proposed method. Thus, we also study the mean and standard deviation of the relative L_2 error for each state variable, along with studying the violin plot of the relative L_2 error for each state variable. We would also like to mention that, unless otherwise stated, we calculate all the errors in the physical space, and not by using the normalized data.

4 Computational experiment: Syngas problem

The applicability and accuracy of the methods proposed in this study are evaluated by considering computational examples. The first computational example considered is the syngas (synthetic gas) problem involving 11 species and 21 reactions [44]. Syngas is a mixture of carbon monoxide (CO) and hydrogen (H_2), and often, carbon dioxide (CO_2). It has various applications, including usage as a fuel or an intermediate for producing chemicals. The total number of state variables for the syngas problem is 12 (i.e, temperature and 11 species' mass fractions).

To train the proposed DeepONet, we generated samples of the dynamics of the system with different initial conditions of the state variables. We considered 51 initial temperature values uniformly spaced between 1000 and 1500 K. The initial conditions for the species' mass fractions are generated from equivalence ratios ranging from 0.7 to 1.3. We sampled 51 equivalence ratios uniformly spaced within the aforementioned range. Thus, we generated a total of $51 \times 51 = 2601$ different samples of initial conditions of the system, i.e., initial temperature and initial species mass fractions. We solved the ODEs (Eq. (1)) of the syngas problem using Cantera's [45] implicit solver with the different initial conditions generated. The solution is obtained on a uniform time step of $\Delta t = 10^{-7}$ s. Thus, we generated a total of 2601 sets of time series of the state variables with different initial conditions. We consider 2080 randomly sampled time series for training, and 520 non-overlapping testing samples.

The chemical process is a continuous-autonomous process, and our dataset starts from $t_0 = \Delta t = 10^{-7}$ s to $t_{end} = 10^4 \Delta t = 10^{-3}$ s. The total number of time steps is 10^4 with a uniform $\Delta t = 10^{-7}$ s. Without loss of generality, we translate the origin from zero to Δt. Thus, the time series data describing the dynamics of the species lie within the temporal range $t = [0, 10^4 - 1]\Delta t$. As discussed in Sections 2 and 3, for the operator learning, we consider a time decomposition. We divide the time series into 99 segments with each segment of length $100 \times \Delta t$ s (i.e., consisting of $n_t + 1 = 101$ time points) and assume that the initial conditions are known for each segment. We would also like to mention that in the time discretization, we have considered the last time point of a segment as the first time point of the next segment, with known initial conditions for each segment (generally coming from the CFD solver). The number of training and testing samples considered is 2080 and 520, respectively, which becomes $2080 \times 99 = 205,920$ and $520 \times 99 = 51,480$, respectively, after performing time decomposition.

We observed the training data set for the syngas problem; the temperature and mass fraction of the species have different orders of magnitude. The temperature is of the order 10^3, and the mass fraction is in the smallest order of 10^{-17}. The minimum and maximum values for the state variables corresponding to the training and testing datasets are shown in Table C.1 (in Appendix C.1). In order to get a better accuracy of the DeepONet prediction, we normalized the output data as follows: we take the logarithm of the state variable and after that normalize each of the state variables separately between $[-1, 1]$ using

$$y_{\text{Norm}} = 2 \times \left(\frac{y - y_{\min}}{y_{\max} - y_{\min}} \right) - 1. \tag{31}$$

where y and y_{Norm} are the values of the state variable before and after the normalization, $y_{\max}$ and $y_{\min}$ are the maximum and value of the state variable (each state variable individually after taking the logarithm) respectively. Similarly, the inputs to the branch network are also normalized. The input to the trunk network is $t = 0, \Delta t, 2\Delta t \ldots, 100\Delta t$, where $\Delta t = 10^{-7}$ s, which is normalized to $[-1, 1]$ using $t_0 = 0$ and $t_{n_t} = 100\Delta t$. In this case, we consider $n_t + 1 = 101$.

In the previous section, we discussed the proposed DeepONet methods for predicting the dynamics of stiff chemical systems with two different strategies for training: (i) the conventional one-step training where both the trunk and branch networks are trained together, (ii) two-step training where the DeepONet is trained sequentially in

two steps. We first discuss the training and the results for the standard DeepONet with one-step training, and then discuss the training and the results for two-step training. As discussed earlier, we consider two types of DeepONet architectures. In the first one, we consider ResNet as the choice of network architecture for both the trunk and branch networks, and call it DON. In the second one, we consider ResNet as the choice of network architecture for the branch and KAN as the choice of network architecture for the trunk, and call it DOK. The details of the network (size, activation function, etc.) considered are shown in Table C.2 (in Appendix C.1).

4.1 Prediction using DeepONet for syngas problem

In this section, we will discuss the training and results of DeepONet for the syngas problem when trained using the one-step training method of DeepONet (discussed in Section 3.1). As discussed, we have normalized the output to $[-1, 1]$. We observed that both the training and testing dataset lies between -1 and 1 after normalization. Thus, in the case of one-step training, we restricted the output of DeepONet as,

$$\widehat{\boldsymbol{Y}}\,(\boldsymbol{\Theta}, \boldsymbol{\Phi}) = 1.05 \times \tanh(\widehat{\boldsymbol{Y}}\,(\boldsymbol{\Theta}, \boldsymbol{\Phi})). \tag{32}$$

We know that tanh( ) is bounded in $(-1, 1)$. Thus, in order to account for the exclusion of the limit, i.e., -1 and 1, we consider the multiplication factor 1.05. Furthermore, 1.05 also acts as a limit for how much extrapolation we expect in case we need to predict extrapolated data. We implemented the PoU in the DeepONet and incorporated loss due to conservation of mass in the loss function as discussed in Section 3.2. We trained the DeepONet with the three different loss functions discussed earlier. First, the standard MSE loss function with additional loss due to conservation of mass (ref. Eqs. (7), (10) and (11)). In this case, we consider the weighting coefficient for $\mathcal{L}_{\text{CoM}}$ ($\mathcal{W}_{\text{CoM}}$) in Eq. (11) as 0.1. The second loss function is the adaptive loss function "Type-A" given by Eq. (12) with $\mathcal{R}_1 = j = 12$. As discussed earlier, we initialized each weight as $\mathcal{W}_a = 1$. In this case, we consider the weight ($\mathcal{W}_{\text{CoM}}$) in Eq. (11) as the 0.1 times the sum of the weights of the data loss, i.e., $\mathcal{W}_{\text{CoM}} = 0.1 \times \sum \mathcal{W}_a$. The third loss function is the adaptive loss function "Type-B" given by Eq. (14) with $\mathcal{R}_2 = j \times bs = 12 \times 205,920$, with initial value of each weight as $\mathcal{W}_{ba} = 1$. In this case, we consider the weight ($\mathcal{W}_{\text{CoM}}$) in Eq. (11) as the 0.1 times the sum of the weights of the data loss, i.e., $\mathcal{W}_{\text{CoM}} = 0.1 \times \sum \mathcal{W}_{ba}$. We also like to mention that as we consider mini-batch, the $\mathcal{W}_{\text{CoM}}$ changes with each mini-batch, even in the same epoch. This is because we have calculated $\mathcal{W}_{\text{CoM}}$ for a respective $\mathcal{W}_{ba}$ corresponding to a particular mini-batch. We have updated the self-adaptive weights, $\mathcal{W}_a$ and $\mathcal{W}_{ba}$ after 100 epochs and at every 50 epochs.

We train the DeepONets up to 15,000 epochs using the Adam [32] optimizer in Tensorflow version 2 environment with 64-bit data precision. We consider 60 mini-batches (i.e., each batch size 3432) up to a 5,000 epoch, after that 120 mini-batches (i.e., each batch size 1716) of the branch input. To check the influence of the initialization of the DeepONet parameters on the accuracy, we train the DeepONets with three independent training runs. The mean of relative L_2 error (as discussed in Section 3.7 with $n_t + 1 = 101$) for each of the three different runs and different methods is shown in Table 2. We observed that in the case of DON, the error reduces from approx 0.154% to approx 0.064% when considering adaptive weight in the loss function (i.e., adaptive loss function Type-B). In the case of DOK, the error reduces from approx 0.101% to approx 0.043%. The difference between the mean of relative L_2 error for the three independent runs is very small, showing the robustness of the training and the minimum effect of initialization of the DeepONet parameters. The convergences of the mean of relative L_2 error with epochs for different methods and runs are shown in Fig. C.6 (in Appendix C.1). The three independent runs follow a similar convergence path with epochs.

Table 2: **Training and testing error for Syngas problem.** Relative L_2 error in training and testing for DON (DeepONet) and DOK (DeepKAN) when trained using standard training (one-step) with different loss functions. The second and third rows indicate the neural network architecture considered in the trunk and branch networks, respectively. The network size and other details are shown in Table C.2 (in Appendix C.1). In the fourth column, we have indicated the type of loss function considered for each case. The last two columns show the mean relative L_2 error (%) for training and testing, respectively, for three independent training runs for each case. The convergence of the mean of relative L_2 error with epoch is shown in Fig. C.6 (in Appendix C.1).

One-step training of DeepONet					
Case	Trunk	Branch	Loss function	Relative L_2 error (%) Training	Testing
DON-NA	ResNet	ResNet	Non-Adaptive	0.154, 0.152, 0.154	0.154, 0.151, 0.153
DON-Ad-A	ResNet	ResNet	Adaptive: Type-A	0.077, 0.077, 0.076	0.076, 0.077, 0.076
DON-Ad-B	ResNet	ResNet	Adaptive: Type-B	0.064, 0.062, 0.061	0.064, 0.062, 0.061
DOK-NA	KAN	ResNet	Non-Adaptive	0.099, 0.099, 0.101	0.099, 0.099, 0.101
DOK-Ad-A	KAN	ResNet	Adaptive: Type-A	0.052, 0.051, 0.054	0.052, 0.050, 0.054
DOK-Ad-B	KAN	ResNet	Adaptive: Type-B	0.042, 0.043, 0.042	0.042, 0.043, 0.043

4.2 Prediction using two-step training of DeepONet for syngas problem

We train our DeepONet with the two-step training method discussed in Section 3.4. As discussed earlier, we did not consider the PoU and conservation of mass constraint in the loss function in the case of two-step training. Furthermore, we have considered the same normalization scheme discussed earlier; however, we have not considered

the bound discussed in the one-step straining given by Eq. (32) as we need to separate the trunk and branch during training. There can be many combinations of network types for both branch and trunk networks, along with varied types of loss functions. Therefore, we first train the trunk network with the two proposed network architectures, i.e., ResNet and KAN, and the three different loss functions considered. After that, we train the branch with the best trunk network obtained from the first step. In order to check the effect of initialization on the accuracy, similar to the one-step training, we consider three independent runs in both cases, the trunk training and the branch training. The size of the $\boldsymbol{A}$ tensor (in Eq. (18)) is $12\times95\times(2080\times99)$.

We train the parameters of the trunk ($\boldsymbol{\Phi}$) and $\boldsymbol{A}$ up to 15,000 epochs using the Adam [32] optimizer in Tensorflow version 2 environment with 64-bit data precision. Similar to one-step training, we consider 60 mini-batches (i.e., each batch size 3432) up to 5,000 epochs, after that 120 mini-batches (i.e., each batch size 1716). We want to mention that, in this case, the mini-batches are considered along the batch-size dimension (or the batch-size axis) of the matrix $\boldsymbol{A}$. The mean of relative L_2 error in training for the trunk training for the three different runs is shown in Table 3. The convergence of the mean of the relative L_2 error with epoch for the trunk training is shown in Fig. C.7 (in Appendix C.1). Similarly to the one-step training, in the trunk training of the two-step training, we observed that the error was reduced when we considered the adaptive loss function in both cases, ResNet and KAN. Furthermore, the error in the case of KAN is smaller than that in ResNet. The errors between different runs are small.

Table 3: **Training error in trunk training in the two-step training of DeepOnet for syngas problem.** Mean of relative L_2 error in trunk training for the two different trunk architectures considered. i.e., ResNet and KAN. The second column shows the type of network architecture, and the third column shows the type of loss function considered. The last column shows the mean of the relative L_2 error (%) in three independent training runs. The convergence of the means of relative L_2 error with epoch is shown in Fig. C.7 (in Appendix C.1).

Two-step training of DeepONet: Trunk training			
Case	Trunk	Loss function	Relative L_2 error (%)
ResNet-NA	ResNet	Non-Adaptive	0.1639, 0.1638, 0.1703
ResNet-Ad-A	ResNet	Adaptive: Type-A	0.0541, 0.0542, 0.0545
ResNet-Ad-B	ResNet	Adaptive: Type-B	0.0120, 0.0143, 0.0132
KAN-NA	KAN	Non-Adaptive	0.1499, 0.1527, 0.1505
KAN-Ad-A	KAN	Adaptive: Type-A	0.0281, 0.0309, 0.0287
KAN-Ad-B	KAN	Adaptive: Type-B	0.0081, 0.0079, 0.0088

After training the trunk network with different types of network architecture and different loss functions, we trained the branch using the reconstructed output labeled data ($\boldsymbol{U} = \boldsymbol{R}^*\boldsymbol{A}^*$). We consider three different loss functions, non-adaptive and two types of adaptive loss functions, as discussed in Section 3.4. We considered the best trunk network architecture/model obtained from the first-stage training, i.e., KAN trained using adaptive loss Type-B. In order to check the influence of the initialization of the network parameters on the accuracy, we train the branch with three independent training runs. Furthermore, for each run, we consider different trunk networks (KAN with Adaptive loss Type-B) obtained in the first stage, i.e., run-1 for the branch considers the trunk obtained in run-1 of the trunk network; run-2 for the branch considers the trunk obtained in run-2 of the trunk network, and run-3 for the branch considers the trunk obtained in run-3. In this case, as well, we consider a mini-batch scheme similar to the previous cases. The mean of relative L_2 error for three independent runs is shown in Table 4. Similar to the previous observation, in this case as well, we observed that the error reduces as the adaptive loss function is considered. The convergences of the mean of relative L_2 error with epoch for different cases are shown in Fig. C.8 (in Appendix C.1). In this case as well, we consider mini-batched, and this is discussed in the Fig. C.8 (in Appendix C.1).

Table 4: **Training and testing error for Syngas problem after branch training.** Relative L_2 error in training and testing for branch training in two-step training of DeepONet. The second and third columns show the type of network architecture considered for the branch and the loss function considered, respectively. The last two columns show the mean relative L_2 error (%) in training and testing for three independent training runs. The convergence of the relative L_2 error with epoch is shown in Fig. C.8 (in Appendix C.1).

Two-step training of DeepONet: After branch training.				
Case	Branch	Loss function	Relative L_2 error (%)	
			Training	Testing
2S-NA	ResNet	Non-Adaptive	0.086, 0.084, 0.084	0.086, 0.084, 0.084
2S-Ad-A	ResNet	Adaptive: Type-A	0.044, 0.046, 0.045	0.044, 0.046, 0.045
2S-Ad-B	ResNet	Adaptive: Type-B	0.041, 0.040, 0.041	0.041, 0.040, 0.041

4.3 Results and discussion

We trained two different types of DeepONet, the first one with ResNet as both the branch and trunk networks, the second one with ResNet as the branch and KAN as the trunk network. We trained these DeepONets with three different types of loss functions: non-adaptive and two adaptive loss functions. We also trained them with two

different training paradigms, i.e., one-step training and two-step training. We observed that the two-step training with KAN as trunk and ResNet as branch gives the smallest mean of the relative L_2 error when trained using adaptive loss function Type-B. However, only one single number (mean in this case) does not provide an accurate synopsis of the methods. Thus, we study the results with the test dataset in more detail. For this, we will study the statistics of the error in more detail. Note that the test dataset consists of 520 samples. As discussed earlier, we had divided these into 99 segments of $100\Delta t$ length, i.e., $n_t + 1 = 101$. We know the initial conditions of each of the 99 segments, and the DeepONet predicts the $100\Delta t$ s advance when the initial conditions are given. In order to study the dynamics, we reconstructed the time series from the DeepONet prediction, i.e., we flattened the time series to a length equal to $99 * 101$. We would like to mention that there are predictions at common time points, as discussed earlier. We calculate the relative L_2 error using Eq. (30) for each state variable and 520 samples (assuming $n_t + 1 = 99 * 101$). The mean and standard deviation of the relative L_2 error for each state variable and method are shown in Table 5. The violin plots of the relative L_2 error for DON-NA, DON-Ad-B, DOK-Ad-B, and 2S-Ad-B for each of the state variables are shown in Fig. 5 (violin plots for all the methods considered are shown in Fig. C.1 (in Appendix C.1)). We observed that state variables describing the dynamics of HO_2 and HCO have the higher mean and standard deviation of errors compared to the rest of the state variables, as seen in Table 5. As expected, the mean and standard deviation of error reduce as we consider the adaptive loss function. Though the overall error in the case of DOK-Ad-B and 2S-Ad-B is nearly the same, the 2S-Ad-B has a smaller standard deviation as observed in Fig. 5, particularly for O, H, HO_2, and HCO. However, the training time for two-step training is higher than the standard one-step training. The training time for different cases is discussed in Section 7. A representative sample dynamics from the test dataset is shown in Fig. 6 and two additional sample results are shown in Fig. C.2 (in Appendix C.1) when predicted using 2S-Ad-B. We observed that the predicted dynamics of the state variables show good accuracy with the true dynamics.

Table 5: **Syngas problem, mean and standard deviation of the % relative L_2 error** for the test samples. The state variables describing the dynamics of HO_2 and HCO have the maximum mean relative L_2 error. We observed that the errors in these two variables were reduced when the adaptive loss functions were considered. Furthermore, the standard deviation of the relation L_2 error is also reduced when adaptive loss functions are considered. The violin plots for the relative L_2 error are shown in Figs. 5 and C.1. The relative L_2 errors are calculated as discussed in Section 4.3.

		DeepONet			DeepOKAN			Two-step training		
State variable		DON-NA	DON-Ad-A	DON-Ad-B	DOK-NA	DOK-Ad-A	DOK-Ad-B	2S-NA	2S-Ad-A	2S-Ad-B
T	μ	0.013	0.014	0.011	0.009	0.010	0.008	0.006	0.007	0.006
	σ	0.005	0.006	0.004	0.003	0.004	0.002	0.002	0.002	0.002
H_2	μ	0.340	0.196	0.099	0.176	0.121	0.069	0.099	0.081	0.052
	σ	0.267	0.118	0.037	0.103	0.069	0.024	0.034	0.032	0.013
O_2	μ	0.108	0.097	0.045	0.072	0.060	0.031	0.044	0.038	0.026
	σ	0.063	0.054	0.013	0.036	0.026	0.009	0.017	0.015	0.007
O	μ	0.482	0.222	0.151	0.319	0.180	0.115	0.203	0.114	0.088
	σ	0.222	0.098	0.047	0.143	0.080	0.040	0.062	0.031	0.021
OH	μ	0.207	0.098	0.090	0.139	0.069	0.058	0.107	0.051	0.054
	σ	0.077	0.039	0.022	0.047	0.026	0.019	0.034	0.017	0.013
H_2O	μ	0.351	0.136	0.097	0.175	0.089	0.066	0.147	0.070	0.066
	σ	0.150	0.051	0.031	0.062	0.035	0.021	0.047	0.025	0.017
H	μ	0.785	0.350	0.201	0.487	0.255	0.139	0.293	0.161	0.116
	σ	0.244	0.138	0.057	0.162	0.097	0.041	0.084	0.046	0.027
HO_2	μ	2.443	0.972	0.432	1.322	0.631	0.280	0.880	0.451	0.250
	σ	1.012	0.450	0.117	0.578	0.264	0.073	0.254	0.151	0.056
CO	μ	0.061	0.055	0.034	0.044	0.039	0.025	0.028	0.025	0.018
	σ	0.023	0.019	0.011	0.018	0.017	0.008	0.009	0.007	0.005
CO_2	μ	0.055	0.036	0.033	0.038	0.026	0.024	0.093	0.043	0.049
	σ	0.021	0.018	0.011	0.015	0.012	0.009	0.032	0.015	0.013
HCO	μ	2.197	0.654	0.305	0.902	0.413	0.205	0.499	0.261	0.182
	σ	1.250	0.355	0.085	0.413	0.189	0.060	0.138	0.071	0.044
N_2	μ	0.002	0.007	0.004	0.002	0.005	0.003	0.001	0.002	0.002
	σ	0.002	0.007	0.003	0.002	0.005	0.002	0.000	0.001	0.001

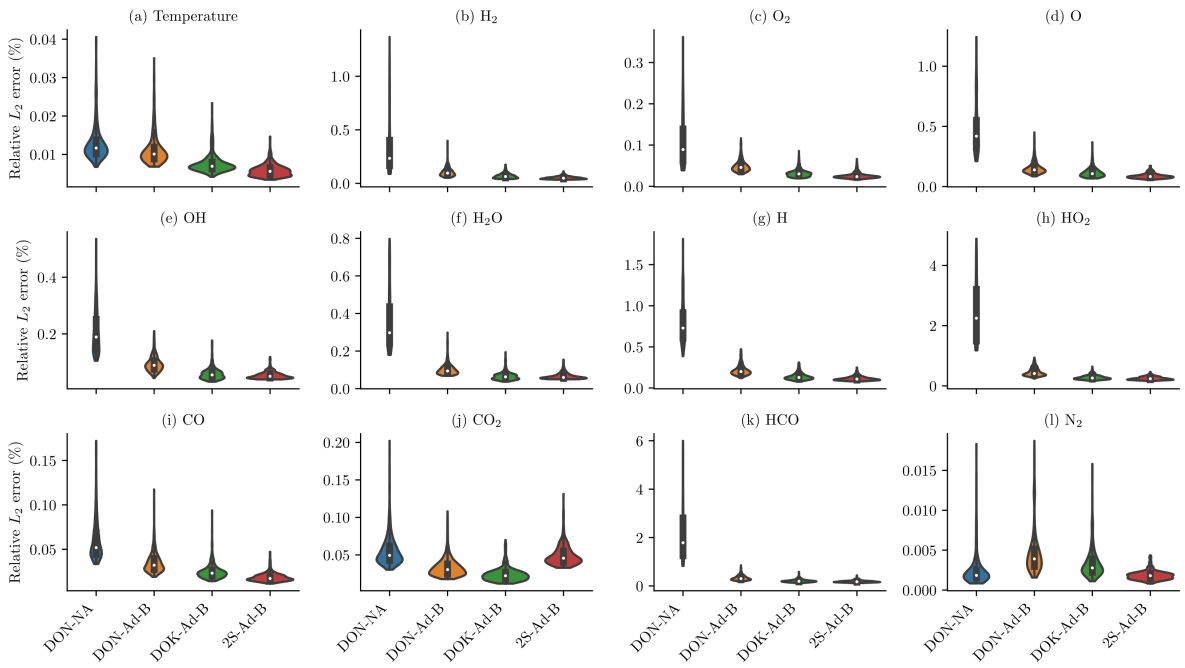
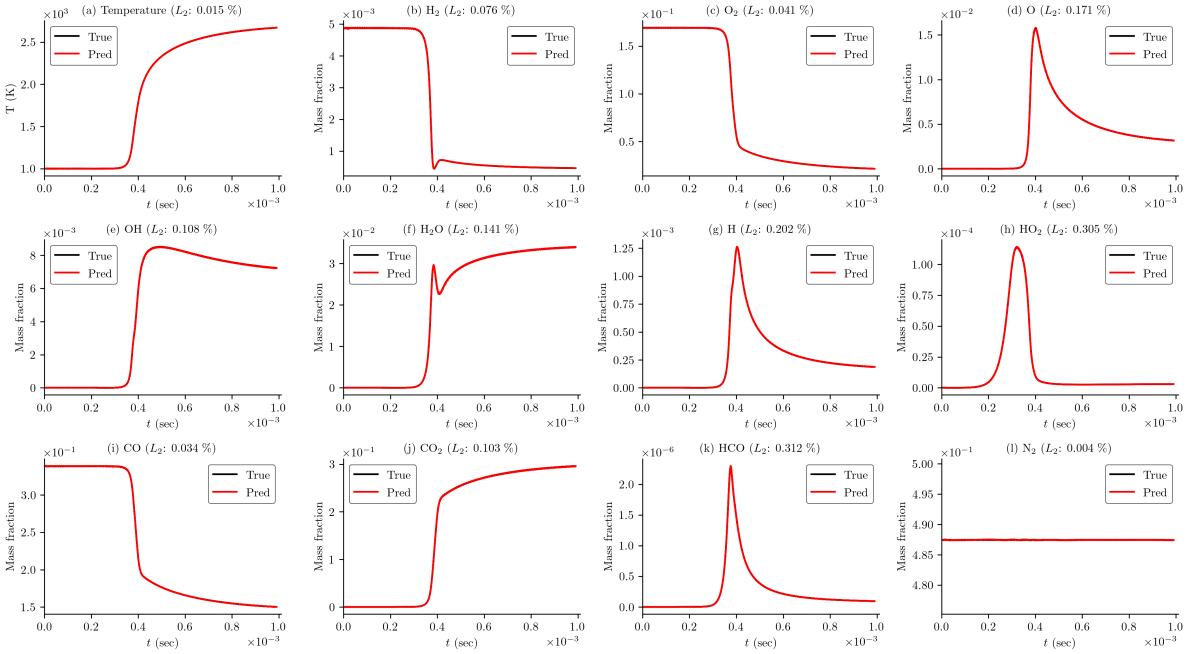


Fig. 5: **Syngas problem, Violin plot** of the relative L_2 error of the reconstructed prediction of the state variables of the test dataset. Here, we have shown only four methods; the violin plots for all the methods considered are shown in Fig. C.1 (in Appendix C.1). We observed that the predicted results using DON-NA have the highest mean of the relative L_2 error compared to the other methods. Furthermore, DON-NA also has a higher standard deviation of the relative L_2 error. The mean and standard deviation are reduced as adaptive loss functions are considered. We would particularly like to mention that the state variables corresponding to the dynamics of HCO and HO_2 shown in (k) and (h), respectively, have higher errors, which are reduced when we consider the adaptive loss function. We have observed that the DON-Ad-B and 2S-Ad-B have similar accuracy; however, 2S-Ad-B has smaller standard deviations. The relative L_2 errors are calculated as discussed in Section 4.3.



(a) Test Sample 1 (2S-Ad-B)

Fig. 6: **Syngas problem, sample test result, 2S-Ad-B:** Plot showing a representative sample result from the test dataset when predicted using 2S-Ad-B, i.e., DeepOKAN trained using the two-step training method and adaptive loss function Type-B. The number in the bracket indicates the relative L_2 errors for each species (calculated as discussed in Section 4.3). The sample result corresponds to one of the higher errors in prediction. The predicted sample shows good accuracy with the true value. Two additional sample results are shown in Fig. C.2 (in Appendix Appendix C.1)

4.4 Mass conserving DeepONet

In the previous sections, we have discussed the results of the proposed DeepONets with different loss functions and training strategies. We have considered a loss function that weakly incorporates conservation of mass in the single-step training strategy; however, this does not ensure that the DeepONet outputs always follow the conservation of mass. In Section 3.5, we have proposed an operator that automatically satisfies the conservation of mass of the species. In this section, we will discuss the implementation and results of the mass-conserving DeepONet. In this case, we consider DeepOKAN (DOK) trained with the adaptive loss function Type-B and trained using two-step training. The number of species for the syngas problem is 11, which, after transformation using forward mapping (ref. Eqs. (B.6) and (B.7)), reduces to 10. Thus, the number of reduced state variables is 11 (temperature and 10 mass fractions of the reduced species). We have considered a similar normalization scheme for the input and output data, as discussed earlier (11 transformed state variables).

We have considered the trunk with KAN architecture with the same network size as considered in Table C.2 (in Appendix C.1), except for the output layer. The number of neurons in the output layer is $11 \times 95 = 1045$ instead of 1140. Similarly, we consider the branch with ResNet architecture with the same network size as considered in Table C.2, except the input layer is 11 instead of 12 and the output layer is $11 \times 95 = 1045$ instead of 1140. The size of the $\boldsymbol{A}$ tensor (in Eq. (18)) is $11 \times 95 \times (2080 \times 99)$. We named this DeepONet as 2S-Ad-B-CoM. We train the parameters of the trunk ($\boldsymbol{\Phi}$) and $\boldsymbol{A}$ up to 15,000 epochs. The mean of the relative L_2 error in training for the trunk training is 0.0080%, 0.0083%, and 0.0081% in the transformed coordinate system, respectively (not normalized), for three independent runs. After the trunk training, we train the branch using the reconstructed labeled data, similar to established procedures discussed earlier. The mean of the relative L_2 error in training is 0.0344%, 0.0362%, 0.0344%, and testing is 0.0344%, 0.0362%, 0.0344% after the branch training for three independent runs in the transformed coordinate system. The predicted output after training is converted back to the physical system using inverse mapping discussed in Section 3.5 and Appendix B.3. The error in the original (physical) system in training is 0.0354, 0.0370, 0.0342, and in testing is 0.0355, 0.0370, 0.0342, respectively. For comparison, we have shown the training and testing error in Table C.3 (in Appendix C.1). We also study the mean and standard deviation of each state variable, similar to the discussion in Section 4.3. The mean and standard deviation of each state variable are shown in Table 6 along with results of 2S-Ad-B. The violin plots for the reconstructed results for the test dataset for the state variable (original) are shown in Fig. 7. We observed that the results of 2S-Ad-B-CoM are marginally better than the results of 2S-Ad-B. We would also like to mention that while in the mass-conserving DeepONet, the sum of the species mass fraction is one, the 2S-Ad-B does not follow conservation of mass. The minimum and maximum values of conservation of mass in the case of 2S-Ad-B in the test dataset are 0.99852 and 1.00162, respectively. Furthermore, we want to mention that additional computation is required in the case of mass-conserving DeepONet for the inverse mapping. The convergence of relative L_2 error with epoch is shown in Fig. C.9 (in Appendix C.1).

Table 6: **Syngas problem, mass conserving DeepONet, mean and standard deviation of the % relative L_2 error** for the test samples when predicted using mass-conserving DeepONet, i.e., 2S-Ad-B-CoM. For comparison, we have also shown the mean and standard deviation of the % relative L_2 error for the test sample when predicted 2S-Ad-B. The violin plots of the relative L_2 error of 2S-Ad-B-CoM and 2S-Ad-B are shown in Fig. 7. The relative L_2 errors are calculated similarly to those discussed in Section 4.3.

Case		T	H_2	O_2	O	OH	H_2O	HO_2	H	CO	CO_2	HCO	N_2
2S-Ad-B	μ	0.006	0.052	0.026	0.088	0.054	0.066	0.116	0.25	0.018	0.049	0.182	0.002
	σ	0.002	0.013	0.007	0.021	0.013	0.017	0.027	0.056	0.005	0.013	0.044	0.001
2S-Ad-B -CoM	μ	0.004	0.048	0.025	0.07	0.045	0.051	0.086	0.178	0.019	0.04	0.144	0.013
	σ	0.001	0.013	0.007	0.017	0.01	0.011	0.022	0.047	0.003	0.009	0.035	0.003

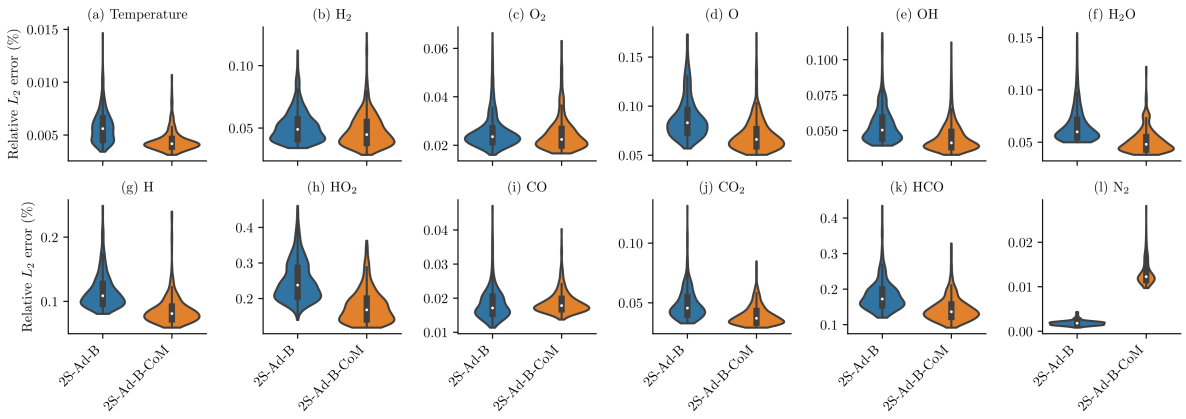
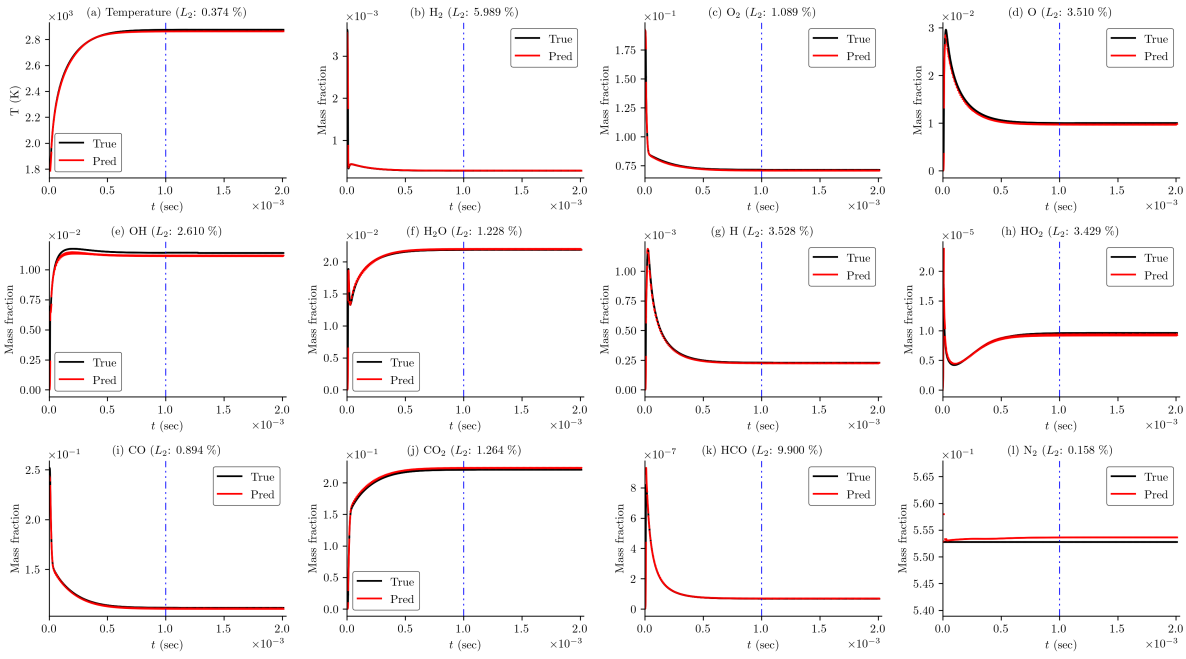


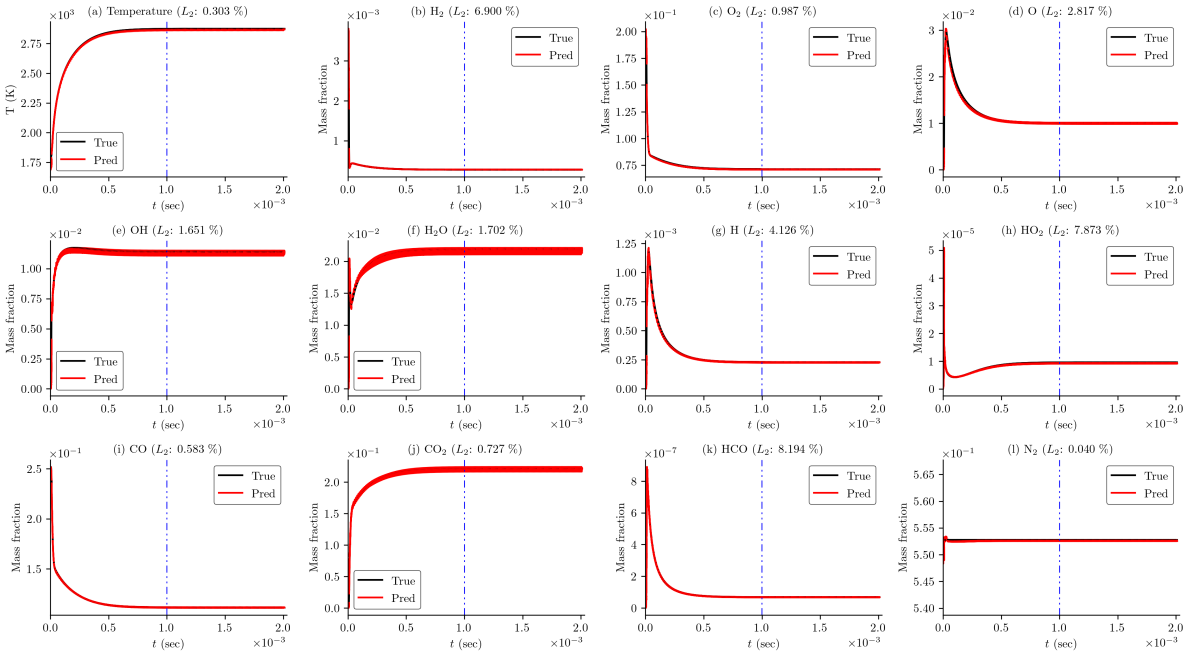
Fig. 7: **Syngas problem, mass conserving DeepONet, Violin plot** for relative L_2 error of the reconstructed prediction of the state variables of the test dataset when predicted using the mass conserving DeepONet (2S-Ad-B-CoM) and compared with 2S-Ad-B discussed in Section 4.2. The relative L_2 errors are calculated similarly to those discussed in Section 4.3.

4.5 Test for extrapolation: Extrapolation using trained DeepONet

In the previous sections, we have discussed the results of the syngas problem predicted using DeepONet, where the test dataset is within the training distribution. In this section, we will test our proposed model on extrapolated data. As mentioned earlier, the training-testing dataset is generated with an initial temperature range of 1000 K to 1500 K and an equivalence ratio range of [0.7, 1.3]. We generated two new datasets with (i) an initial temperature of 1800 K and an equivalence ratio of 0.9, (ii) an initial temperature of 1800 K and an equivalence ratio of 1.5. We predict the results using the trained model DOK-Ad-B and 2S-Ad-B in the previous sections, assuming the initial conditions are known for each time segment. The first set of results is shown in Fig. 8 and the second set of results in Fig. C.3 (in Appendix C.1). In the case of the first dataset, we also tested our model beyond the training time zone. The training time zero is 0 to $\sim 1\times10^{-3}$ s. We tested up to $\sim 2\times10^{-3}$ s. The results show good accuracy. In the second dataset, we observed that the prediction using DOK-Ad-B is marginally poor. Particularly, the predicted maximum output is smaller than the true values. We would like to mention that in the case of DOK-Ad-B, we have restricted our output using $1.05\times\tanh(\,)$ function (ref. Eq. (32)). This restricts the maximum allowable extrapolation. We can modify the 1.05 and allow maximum allowable extrapolation. Furthermore, we also observed in both cases that predictions using 2S-Ad-B are noisier than the predictions using DOK-Ad-B. A rigorous theoretical or mathematical analysis of why the predictions using 2S-Ad-B in these cases are noisier is beyond the scope of the present study. However, we believe that the tanh( ) (ref. Eq. (32)) function in DOK-Ad-B in the output of DeepONet makes the output smoother or less noisy. Furthermore, in the case of DOK-Ad-B, we also consider the PoU in the trunk output.



(a) DOK-Ad-B, Extrapolation sample # 1



(b) 2S-Ad-B, Extrapolation sample # 1

Fig. 8: **Syngas problem, DeepONet extrapolation result # 1.** Plot showing the dynamics of the state variables of the syngas problem for extrapolation dataset #1 when predicted using trained (a) DOK-Ad-B and (b) 2S-Ad-B method. The predicted dynamics show good accuracy with the true dynamics.

4.6 Recursive prediction using DeepONet

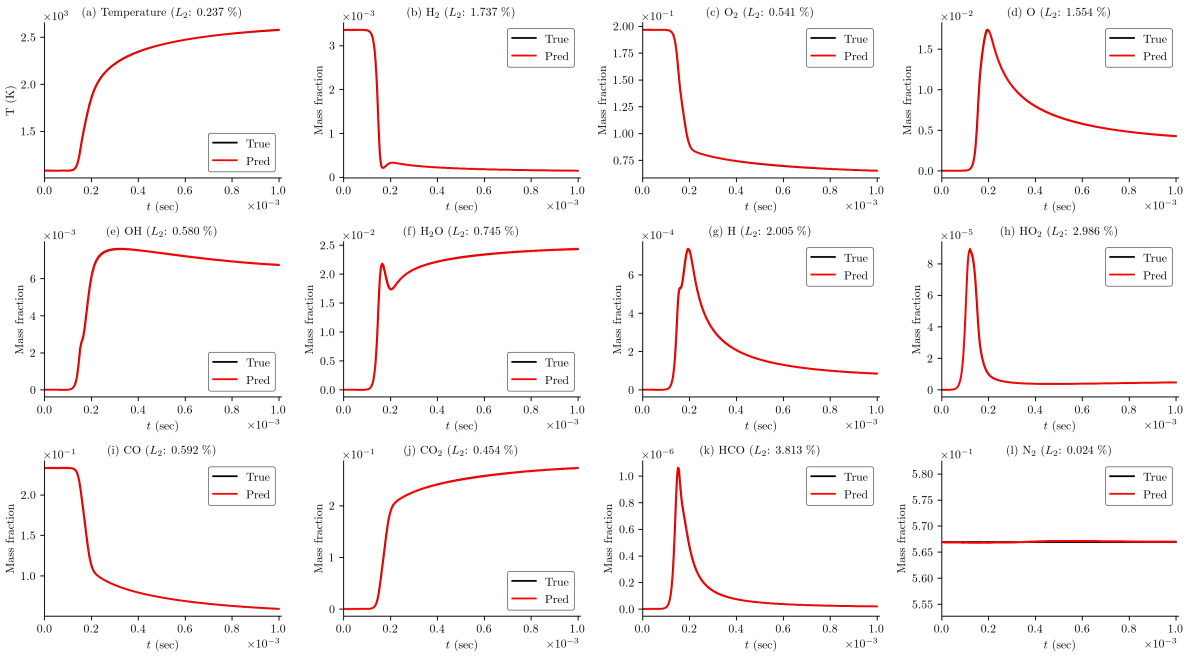
In the previous sections, we have discussed the training process and the results of DeepONets when the initial conditions for each of the time segments are known. As discussed in Section 2, we will test our DeepONet models for long-time prediction using the recursive/auto-regressive prediction method discussed in Section 3.6. For this purpose, we consider the DeepONet models trained in the previous section to predict the output auto-regressively by considering the last output of the DeepONet as the input to predict the dynamics of the second segment, and so on. We consider the DeepONet model 2S-Ad-B trained in the previous section to predict the auto-regressive dynamics and call it 2S-Ad-B-R. In Section 4.4, we also discussed the mass-conserving DeepONet results. We also consider this model for recursive prediction and call it 2S-Ad-B-CoM-R. The mean of test errors for the two methods for the three models (parameters of three independent runs) are shown in Table 7, when calculated using Eq. (30) with $n_t + 1 = 101$. We consider the complete test dataset, i.e., 520 test samples discussed earlier. Similar to Section 4.3, we study the statistics of the relative L_2 error of the reconstructed dynamics. The mean, standard deviation, median, 75 percentile (q:75), 90 percentile (q:90), and maximum error of the relative L_2 error of the reconstructed test samples for 2S-Ad-B-R and 2S-Ad-B-CoM-R are shown in Table 8. The violin plots for L_2 error for the reconstructed output dynamics are shown in Fig. C.4 (in Appendix C.1). We observed that while most of the dynamics show good accuracy, there are high errors in a few of the samples, as long tails are observed in the violin plots. Dynamics of a test samples corresponding to 90 percentile error in HCO is Fig. 9 and additional dynamics corresponds to test sample are shown in Fig. C.5 (in Appendix C.1). The dynamics show good accuracy. The accumulation of error over time for each state variable is shown in Fig. 10.

Table 7: **Syngas problem, testing error for recursive prediction:** Relative L_2 error in prediction for test dataset when recursively predicted the output as discussed in Section 4.6 with $n_t + 1 = 101$, predicted using DeepOKAN trained using Adaptive loss Type-B with two step training method (2S-Ad-B-R) and using mass conserving DeepOKAN trained using Adaptive loss Type-B with two step training method (2S-Ad-B-CoM-R). The table shows the prediction using the three independently trained models for each DeepONet discussed in the previous sections.

Case	Relative L_2 error (%)
2S-Ad-B-R	0.632, 0.509, 606
2S-Ad-B-CoM-R	0.643, 0.565, 0.556

Table 8: **Syngas problem, Recursive prediction**, mean, standard deviation, median, 75 percentile (q:75), 90 percentile (q:90), and maximum error of the relative L_2 error (%) of the reconstructed test samples for 2S-Ad-B-R and 2S-Ad-B-CoM-R. The state variables describing the dynamics of HO_2 and HCO have the maximum mean relative L_2 error. We observed that 90 percentile of all the state variables have an error smaller than 5% relative L_2 error. The relative L_2 errors are calculated similarly to the discussion in Section 4.3. The violin plots of relative L_2 error for each state variable are shown in Fig. C.4 (in Appendix C.1)

States	2S-Ad-B-R						2S-Ad-B-CoM-R					
	μ	σ	Median	q:75	q:90	Max	μ	σ	Median	q:75	q:90	Max
T	0.19	0.36	0.10	0.16	0.29	3.48	0.17	0.37	0.07	0.12	0.32	4.54
H_2	0.64	1.59	0.14	0.31	1.56	13.41	0.72	1.59	0.20	0.39	2.06	15.67
O_2	0.45	0.79	0.24	0.41	0.80	7.44	0.52	0.83	0.30	0.50	0.97	11.17
O	1.08	2.62	0.42	0.68	1.84	24.56	1.13	2.75	0.43	0.67	2.26	35.28
OH	0.49	0.88	0.28	0.47	0.79	8.44	0.48	0.90	0.27	0.41	0.85	11.33
H_2O	0.48	1.16	0.18	0.33	0.79	10.60	0.52	1.17	0.20	0.37	0.98	12.66
H	1.09	2.94	0.31	0.50	2.08	27.02	1.13	3.04	0.30	0.49	2.77	35.83
HO_2	1.52	3.19	0.64	0.97	3.00	28.60	1.45	3.26	0.49	0.74	3.70	37.62
CO	0.30	0.58	0.15	0.26	0.54	5.22	0.38	0.58	0.20	0.38	0.76	5.52
CO_2	0.36	0.82	0.17	0.28	0.61	7.78	0.38	0.83	0.16	0.28	0.69	10.39
HCO	1.73	4.95	0.32	0.63	3.81	43.91	1.84	5.02	0.38	0.70	4.66	52.83
N_2	0.03	0.02	0.03	0.04	0.06	0.14	0.06	0.04	0.04	0.06	0.11	0.30



Recursive prediction, Sample results. 90 percentile error in HCO.

Fig. 9: **Syngas problem, Recursive prediction, sample results.** Plot showing the dynamics of the state variables of the syngas problem for test dataset when predicted using recursive prediction. Plot shows the dynamics of the state variables corresponding to 90 percentile error in predicted state variable HCO (3.81%), The plots show the dynamics of the predicted state variable when predicted recursively using the trained model 2S-Ad-B, which we called 2S-Ad-B-R. The plots show good accuracy in prediction. Additional results are shown in Fig. C.5 (in Appendix C.1) (in Appendix C.1).

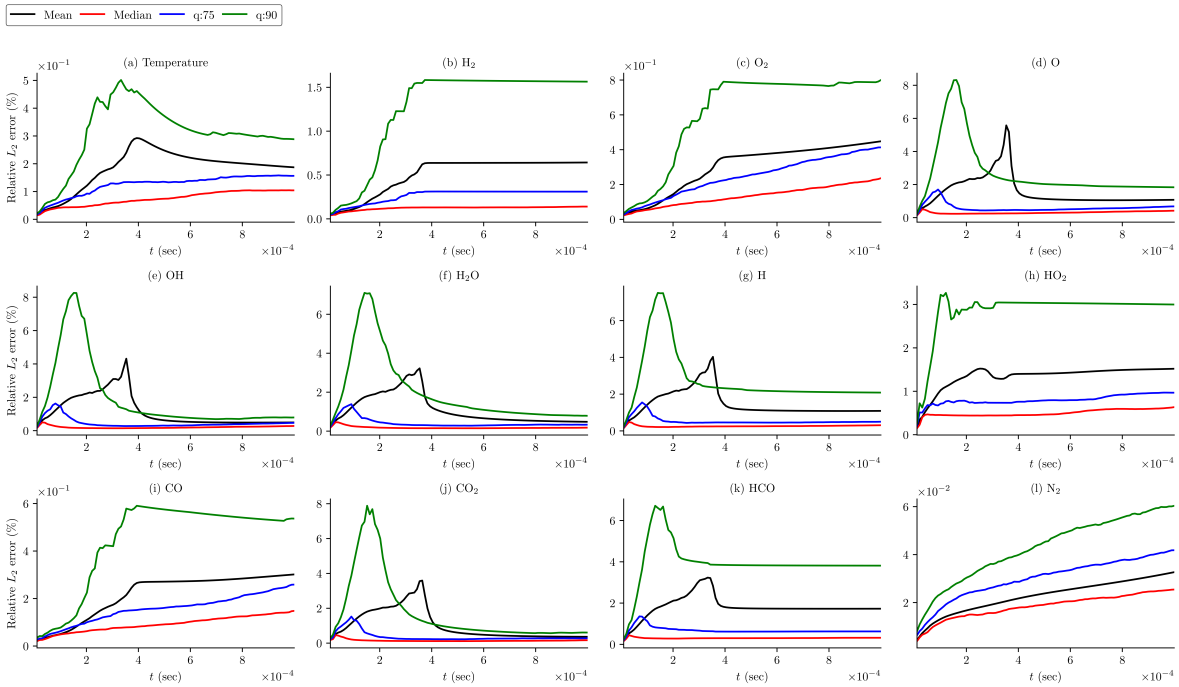


Fig. 10: **Syngas problem, recursive prediction, error accumulation:** Plots show the accumulation of error over time for each state variable when predicted recursively in an autoregressive manner as discussed in Section 4.6. The relative L_2 errors are calculated similarly to the discussion in Section 4.3 using the complete output up to that autoregressive step. Since the error is skewed, we have shown the median, 75 percentile (q:75), and 90 percentile (q:90) error growth along with mean error accumulation for the test dataset. We observed that the mean error is higher than the median and 75 percentile error, indicating a high skewness. From the 90 percentile error, we observed that maximum errors occur in the time zone 1×10^{-4} s to 4×10^{-4} s. The violin plots of the relative error at the end of time are shown in Fig. C.4.

5 Computational experiment: GRI - Mech 3.0 problem

In the previous section, we showed that the proposed DeepONet-based framework AMORE can predict the dynamics of the stiff chemical system corresponding to the Syngas mechanism with good accuracy. In this regard, we shall show the scalability of the proposed framework to a larger number of state variables. To this end, we consider the GRI-Mech 3.0 as our second computational example. The GRI-Mech 3.0 is a detailed chemical reaction mechanism for natural gas combustion. GRI-Mech 3.0 is specifically designed to simulate the combustion of methane (CH_4) and its mixtures with air. The GRI-Mech 3.0 mechanism includes 53 species and 325 reactions, providing a detailed description of combustion processes for methane-based fuels. Similar to the Syngas problem discussed earlier (Section 4), we generate data for training and testing of DeepONet with the initial condition of temperature within the range of 1000 K to 1500 K and the equivalence ratio between 0.7 to 1.3. The system has 54 state variables, i.e., temperature, along with the 53 state variables describing the dynamics of the mass fraction of each of the 53 species. From the simulated dataset, we have observed that out of 54 state variables, only 24 have significant values or non-negative and non-zero values. Thus, we have considered only 24 state variables. The minimum and maximum values of each state variable for the training and testing datasets are shown in Table C.4 (in Appendix C.2). Similar to the syngas problem, we consider 2080 samples for training and 520 samples for testing. The sum of the species mass fraction (23 species) at any time instant lies between 1.0000002 and 0.9999988 for the training dataset and 1.000000013 and 0.9999988 for the testing dataset. Similar to the syngas problem, we consider the time series from $t_0 = 10^{-7}$ s, which after the shifting of the time axis becomes $t = 0$ s. In this computational example, as well, we have divided the time series into 99 segments, each of length $100\Delta t$, s where $\Delta t = 10^{-7}$. Thus, after performing the time decomposition, the number of training samples becomes $2080\times99 = 205920$, and the number of testing samples becomes $520\times99 = 51480$. For the sake of convenience, from now onward, we will refer to the GRI-Mech 3.0 problem as the GRI problem.

5.1 Prediction using DeepONet

In the previous problem (Section 4 for Syngas problem), we have considered different types of DeepONets with different loss functions and training strategies. We observe that DeepOKAN trained using the two-step training method with adaptive loss function Type-B shows better accuracy compared to the other methods. Therefore, for the GRI problem, we only consider DeepOKAN trained using the two-step training method with adaptive loss function Type-B. The network size considered for the GRI problem is shown in Table C.5 (in Appendix C.2). First, we trained the trunk network; in this case, the size of $\boldsymbol{A}$ is $24\times95\times(2080\times99)$. We trained the parameters

of the trunk and $\boldsymbol{A}$ up to 20,000 epochs. The % relative L_2 error in the training of the trunk network for three independent runs are 0.0303, 0.0294, and 0.0288. After the trunk training, we trained the parameters of the branch network up to 20,000 epochs with the reconstructed output similar to the syngas problem discussed earlier. The % relative L_2 error after the branch training for three independent runs are 0.0853, 0.0769 and 0.0847 in training and 0.0837, 0.0754 and 0.0831 in testing, respectively. The convergence of relative L_2 error in trunk training and branch training and testing is shown in Fig. C.10 (in Appendix C.2).

Similar to the discussion in Section 4.3, we study the statistics of the relative L_2 error of the reconstructed output of the testing dataset. The mean and standard deviation of the relative L_2 error for each state variable are shown in Table C.6 (in Appendix C.2), and the violin plot of the relative L_2 error for each state variable is shown in Fig. 11. The results show good accuracy across all state variables. A representative dynamics of the test sample is shown in Fig. 12, two additional representative dynamics of the test sample are shown in Fig. C.11 (in Appendix C.2).

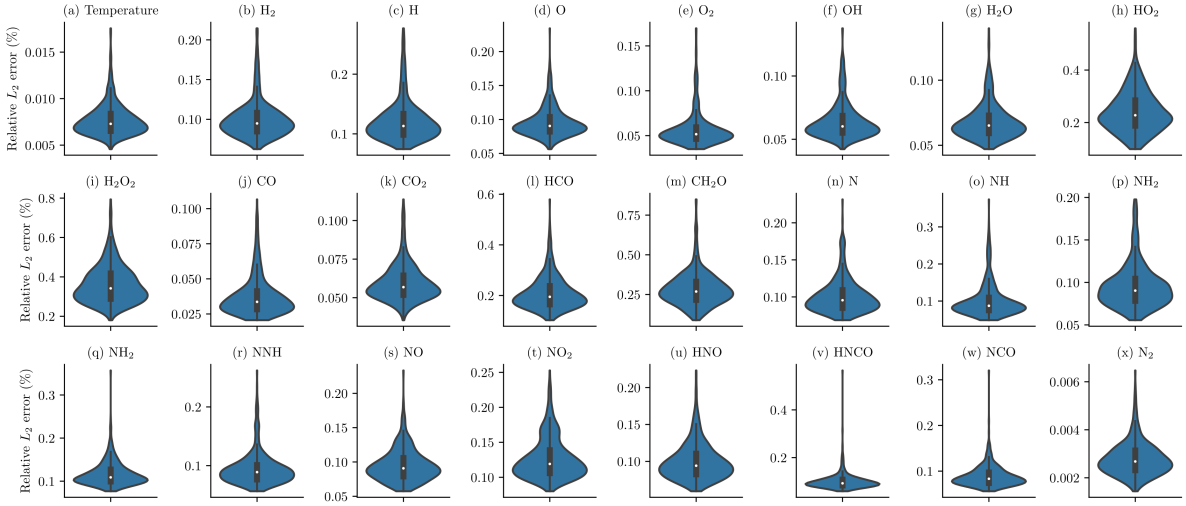
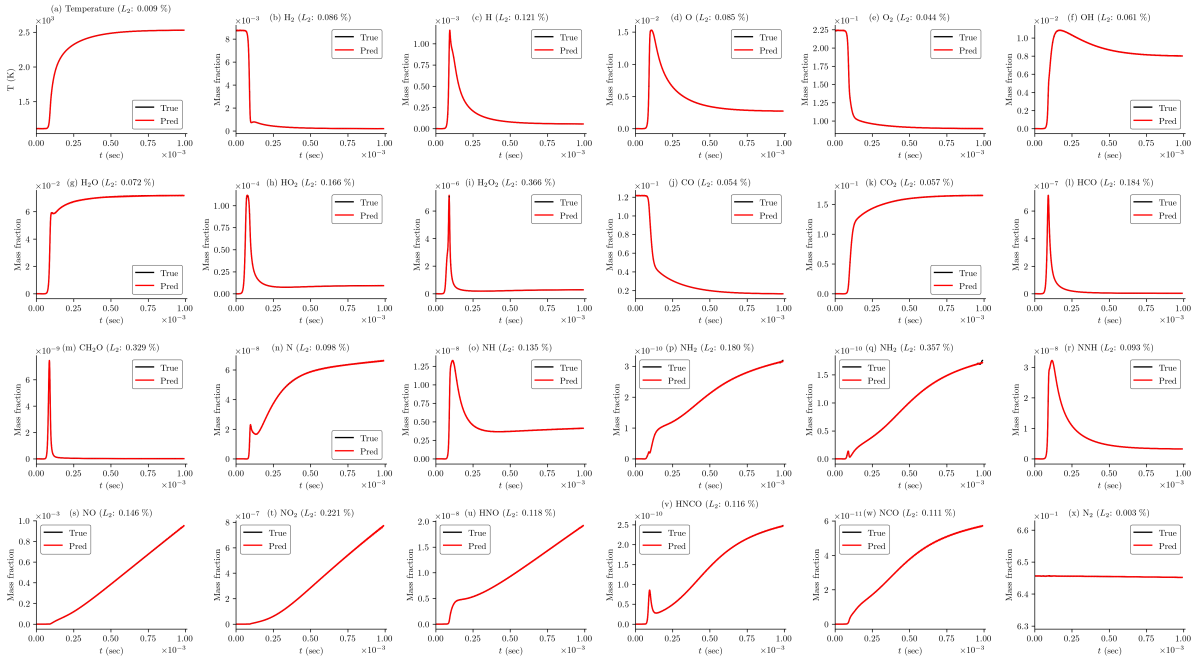


Fig. 11: **GRI problem, Violin plot** of the relative L_2 error of the reconstructed prediction of the state variables of the test dataset. The relative L_2 errors are calculated similarly as discussed in Section 4.3



(a) Sample 1

Fig. 12: **GRI problem, sample result from test dataset.** Plot showing the two sample predicted dynamics of the state variables for GRI problem. The sample result corresponds to one of the higher errors in prediction. The predicted sample shows good accuracy with the true value. Additional sample results are shown in Fig. C.11 (in Appendix C.2).

6 AMORE implementation in Fourier Neural Operator

In the previous sections, we have shown how the proposed AMORE framework can be considered to improve the accuracy of DeepONet with multiple outputs. AMORE is a general framework and can be considered with other neural operators. In this section, we will implement AMORE with FNO [31]. For this purpose, we will approximate the dynamics of the syngas problem discussed earlier using FNO and train using non-adaptive MSE loss function (FNO-NA) given by Eq. (7) and adaptive loss function Type-B (FNO-Ad-B) given by Eq. (14). We highlight that the objective is not to compare the performance of DeepONet and FNO, as the accuracy depends on a combination of lots of factors like network size, training strategies, learning rate scheduler, batch size, and number of epochs. The interested reader may refer to a comparative study by Lu et al. [20].

We consider an FNO for the syngas problem and train it for 3,000 epochs, which takes around 13.5 hrs, while we trained DeepONet for 15,000 epochs, which takes around 10.5 hrs. The mean relative error in training is 0.17% and in testing is 0.17% when considering non-adaptive loss (Eq. (7)). The mean relative error in training reduces to 0.13% and in testing is reduced to 0.13% when trained using adaptive loss function Type-B (Eq. (14)). The convergence of the relative L_2 error in training and testing are shown in Fig. C.13 (in Appendix C.3). As discussed earlier, in Section 4.3, we study the reconstructed test data. The violin plot of the relative L_2 error of the reconstructed prediction of the test dataset for FNO-NA and FNO-Ad-B are shown in Fig. 13. We observed that, similar to DeepONet, the accuracy of FNO is also improved when considering an adaptive loss function Type-B. The mean and standard deviation of the relative L_2 error for the reconstructed test dataset for each state variable is shown in Table C.8 (Appendix C.3). A brief discussion on FNO and its implementation in the present study, including additional results, is included in Appendix C.3

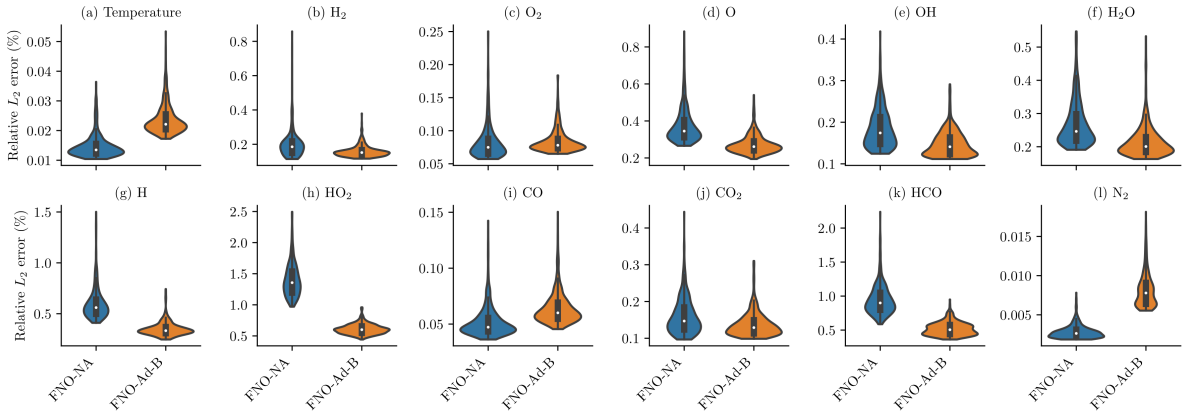


Fig. 13: **Syngas problem, FNO, Violin plot** of the relative L_2 error of the reconstructed prediction of the state variables of the test dataset. We observed that the prediction using the adaptive loss function gives better accuracy. The relative L_2 errors are calculated similarly as discussed in Section 4.3

7 Computational cost

The training of DeepONet is offline, and once trained, it can predict the dynamics of a stiff chemical system. In the present study, we have considered two different types of DeepONet, i.e., DON, where both branch and trunk are ResNet, and DOK, where the branch is ResNet and the trunk is KAN, and two different paradigms of training methods. The training time for each of the methods is discussed in Appendix D. We observed that the training time for the adaptive loss functions is marginally higher than that of the non-adaptive loss function. Furthermore, the training time for the two-step training is higher than for one-step training.

The training of DeepONets is offline, thus it does not affect the prediction time or eventual integration of DeepONet with a CFD solver. As discussed earlier, the mass-conserving DeepONet is computationally more expensive than other DeepONets as it requires the inverse mapping. Future studies may focus on reducing computational time in the inverse mapping. However, it may be noted that when integrating with a CFD solver, it is not necessary to consider inverse mapping at all the time points. One may choose to use the mapping only at the time point where the CFD integration is performed.

8 Summary and Conclusion

We have proposed an adaptive multi-output operator network (AMORE) framework for the accurate prediction of multiple state variables of stiff chemical kinetics from a single neural operator. We consider the Deep Operator Network (DeepONet) to predict all the thermochemical state variables. We consider two DeepONet architectures, the first one with ResNet as the choice of network for both the branch and trunk networks. The second one with ResNet as the choice of network for the branch network and KAN for the trunk network. The proposed DeepONet architectures are capable of predicting the dynamics of multiple state variables from a single DeepONet. We design our trunk network such that the partition of unity (PoU) condition is automatically satisfied. In order to obtain

better accuracy, we proposed two gradient-free adaptive loss functions. The idea of these adaptive loss functions is that the state variables and/or the samples with higher error need to be weighted more in the optimization process. The weights are evaluated based on the error in the previous epoch. We consider the two-step training of DeepONet for multiple output, where each output variable have its individual basis function and coefficients. Furthermore, we also extend the proposed adaptive loss functions to two-step training of DeepONet for both the trunk and branch networks. We know that the total mass of a chemical system is conserved, i.e., the total mass of the reactant and product is the same. We proposed a DeepONet that automatically satisfies the conservation of mass in terms of mass fraction of the species of the system.

We have shown that the proposed DeepONet architectures within our AMORE framework were able to predict the dynamics of the state variables (i.e., temperature and mass fraction) with good accuracy. The computational examples with 12 (Syngas problem) and 24 (GRI-Mech 3.0 problem) output state variables show the accuracy, applicability, and scalability of the proposed method. We have observed that the adaptive weights in the loss function improve the accuracy of the prediction. Furthermore, the DeepONet with KAN as the trunk network provides more reliable predictions with smaller mean and standard deviation of the test samples. The DeepONet trained using two-step training provides marginally better results than single-step training. However, the training time for two-step training is higher than that of single-step training.

We tested our proposed DeepONet for out-of-distribution results and recursive/auto-regressive prediction. The results are within acceptable error limits. In the case of auto-regressive prediction, we considered the complete time series for auto-regressive prediction, which is a highly conservative case. However, even in this case, we observed that the error in the results is less than 5% relative L_2 error for the 90th percentile of the test cases, showing their usefulness as a surrogate for stiff chemical kinetics and potential for a hybrid model for operator-CFD solver.

The proposed AMORE framework can act as a foundation block for future studies of coupled operators and CFD models, where the operator will simulate the solution of the stiff chemical system. The obvious future research step is to combine neural operators with a CFD solver. In this case, future research can also focus on developing a hybrid model [46] that combines the proposed DeepONet and the Physics-Informed Neural Network (PINN) [9]. The DeepONet can act as a surrogate for the stiff chemical kinetics, and PINN acts as a solver for fluid flow. We would also like to mention that the mass-conserving DeepONet is computationally expensive compared to the other proposed DeepONet architectures because of the inverse mapping. Future studies may also consider developing strategies for reducing the computation time for the inverse mapping.

The proposed AMORE framework (with self-adaptive loss functions) can effectively reduce the error for multiple output state variables for stiff chemical kinetics. The methodology and the proposed framework is universal in a way that it can be considered for other systems and operators. We have demonstrated this by considering FNO within the AMORE framework. Future research directions could also focus on reducing the training cost, particularly for two-step training, and implementing it on other systems.

Data availability

Data will be made available upon request to the corresponding author.

Acknowledgement

We want to acknowledge the support from the Small Business Technology Transfer (STTR) program, USA (Grant No: GR5291245). This research was conducted using computational resources and services at the Center for Computation and Visualization, Brown University.

Author Contributions Statement

Kamaljyoti Nath: Conceptualization, Methodology, Validation, Investigation, Visualization, Software, Result generation and analysis, Writing- original draft, Writing – review & editing; **Additi Pandey**: Conceptualization, Software, Validation, Writing- original draft, Writing – review & editing; **Bryan T. Susi**: Conceptualization, Supervision, Project administration, Writing – review & editing; **Hessam Babaee**: Conceptualization, Methodology, Data curation, Supervision, Writing - original draft, Writing – review & editing; **George Em Karniadakis**: Conceptualization, Methodology, Funding acquisition, Project administration, Resources, Supervision, Writing - original draft, Writing – review & editing. All authors reviewed the manuscript.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] U. Maas and S. B. Pope. Simplifying chemical kinetics: Intrinsic low-dimensional manifolds in composition space. *Combustion and Flame*, 88(3):239–264, 1992. doi: https://doi.org/10.1016/0010-2180(92)90034-M. URL https://www.sciencedirect.com/science/article/pii/001021809290034M.

[2] V. Bykov and U. Maas. The extension of the ildm concept to reaction–diffusion manifolds. *Combustion Theory and Modelling*, 11(6):839–862, 11 2007. doi: 10.1080/13647830701242531. URL https://doi.org/10.1080/13647830701242531.

[3] S. H. Lam and D. A. Goussis. The csp method for simplifying kinetics. *International Journal of Chemical Kinetics*, 26(4):461–486, 2021/06/10 1994. doi: https://doi.org/10.1002/kin.550260408. URL https://doi.org/10.1002/kin.550260408.

[4] K. S. Jung, C. E. Lacey, H. Babaee, and J. H. Chen. Accelerating high-fidelity simulations of chemically reacting flows using reduced-order modeling with time-dependent bases. *Computer Methods in Applied Mechanics and Engineering*, 437:117758, 2025. doi: https://doi.org/10.1016/j.cma.2025.117758. URL https://www.sciencedirect.com/science/article/pii/S0045782525000301.

[5] Alisha J Sharma, Ryan F Johnson, David A Kessler, and Adam Moses. Deep learning for scalable chemical kinetics. In *AIAA scitech 2020 forum*, page 0181, 2020.

[6] Thomas S. Brown, Harbir Antil, Rainald Löhner, Fumiya Togashi, and Deepanshu Verma. Novel DNNs for Stiff ODEs with Applications to Chemically Reacting Flows. In Heike Jagode, Hartwig Anzt, Hatem Ltaief, and Piotr Luszczek, editors, *High Performance Computing*, pages 23–39, Cham, 2021. Springer International Publishing. ISBN 978-3-030-90539-2.

[7] Mario De Florio, Enrico Schiassi, and Roberto Furfaro. Physics-informed neural networks and functional interpolation for stiff chemical kinetics. *Chaos (Woodbury, N.Y.)*, 32, 06 2022. doi: 10.1063/5.0086649.

[8] Matthew Frankel, Mario De Florio, Enrico Schiassi, and Lina Sela. Hybrid chemical and data-driven model for stiff chemical kinetics using a physics-informed neural network. *Engineering Proceedings*, 69(1):40, 2024.

[9] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[10] Enrico Schiassi, Roberto Furfaro, Carl Leake, Mario De Florio, Hunter Johnston, and Daniele Mortari. Extreme theory of functional connections: A fast physics-informed neural network method for solving ordinary and partial differential equations. *Neurocomputing*, 457:334–356, 2021.

[11] Weiqi Ji, Weilun Qiu, Zhiyu Shi, Shaowu Pan, and Sili Deng. Stiff-PINN: Physics-Informed Neural Network for Stiff Chemical Kinetics. *The Journal of Physical Chemistry A*, 125(36):8098–8106, 2021. doi: 10.1021/acs.jpca.1c05102. URL https://doi.org/10.1021/acs.jpca.1c05102. PMID: 34463510.

[12] Simone Venturi and Tiernan Casey. SVD perspectives for augmenting DeepONet flexibility and interpretability. *Computer Methods in Applied Mechanics and Engineering*, 403:115718, 2023. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2022.115718. URL https://www.sciencedirect.com/science/article/pii/S0045782522006739.

[13] Lu Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang, and George Em Karniadakis. Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. *Nature Machine Intelligence*, 3(3):218–229, mar 2021. ISSN 2522-5839. doi: 10.1038/s42256-021-00302-5. URL http://dx.doi.org/10.1038/s42256-021-00302-5.

[14] Somdatta Goswami, Ameya D. Jagtap, Hessam Babaee, Bryan T. Susi, and George Em Karniadakis. Learning stiff chemical kinetics using extended deep neural operators. *Computer Methods in Applied Mechanics and Engineering*, 419:116674, 2024. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2023.116674. URL https://www.sciencedirect.com/science/article/pii/S0045782523007971.

[15] Anuj Kumar and Tarek Echekki. Combustion chemistry acceleration with DeepONets. *Fuel*, 365:131212, 2024. ISSN 0016-2361. doi: https://doi.org/10.1016/j.fuel.2024.131212. URL https://www.sciencedirect.com/science/article/pii/S0016236124003582.

[16] Yuting Weng, Han Li, Hao Zhang, Zhi X. Chen, and Dezhi Zhou. Extended Fourier Neural Operators to learn stiff chemical kinetics under unseen conditions. *Combustion and Flame*, 272:113847, 2025. ISSN 0010-2180. doi: https://doi.org/10.1016/j.combustflame.2024.113847. URL https://www.sciencedirect.com/science/article/pii/S001021802400556X.

[17] G. E. P. Box and D. R. Cox. An analysis of transformations. *Journal of the Royal Statistical Society: Series B (Methodological)*, 26(2):211–243, 07 1964. ISSN 0035-9246. doi: 10.1111/j.2517-6161.1964.tb00553.x. URL https://doi.org/10.1111/j.2517-6161.1964.tb00553.x.

[18] Tianping Chen and Hong Chen. Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems. *IEEE Transactions on Neural Networks*, 6(4):911–917, 1995.

[19] Pengzhan Jin, Shuai Meng, and Lu Lu. MIONet: Learning Multiple-Input Operators via Tensor Product. *SIAM Journal on Scientific Computing*, 44(6):A3490–A3514, 2022. doi: 10.1137/22M1477751. URL https://doi.org/10.1137/22M1477751.

[20] Lu Lu, Xuhui Meng, Shengze Cai, Zhiping Mao, Somdatta Goswami, Zhongqiang Zhang, and George Em Karniadakis. A comprehensive and fair comparison of two neural operators (with practical extensions) based on FAIR data. *Computer Methods in Applied Mechanics and Engineering*, 393:114778, 2022. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2022.114778. URL https://www.sciencedirect.com/science/article/pii/S0045782522001207.

[21] Lizuo Liu, Kamaljyoti Nath, and Wei Cai. A Causality-DeepONet for Causal Responses of Linear Dynamical Systems. *Communications in Computational Physics*, 35(5):1194–1228, 2024. ISSN 1991-7120. doi: https://doi.org/10.4208/cicp.OA-2023-0078. URL http://global-sci.org/intro/article_detail/cicp/23189.html.

[22] Min Zhu, Shihang Feng, Youzuo Lin, and Lu Lu. Fourier-DeepONet: Fourier-enhanced deep operator networks for full waveform inversion with improved accuracy, generalizability, and robustness. *Computer Methods in Applied Mechanics and Engineering*, 416:116300, 2023. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2023.116300. URL https://www.sciencedirect.com/science/article/pii/S0045782523004243.

[23] Elham Kiyani, Manav Manav, Nikhil Kadivar, Laura De Lorenzis, and George Em Karniadakis. Predicting crack nucleation and propagation in brittle materials using Deep Operator Networks with diverse trunk architectures. *Computer Methods in Applied Mechanics and Engineering*, 441:117984, 2025. ISSN 0045-7825. doi: 10.1016/j.cma.2025.117984.

[24] Peter Rivera-Casillas, Sourav Dutta, Shukai Cai, Mark Loveland, Kamaljyoti Nath, Khemraj Shukla, Corey Trahan, Jonghyun Lee, Matthew Farthing, and Clint Dawson. A Neural Operator-Based Emulator for Regional Shallow Water Dynamics. *arXiv pre-print. 2502.14782*, 2025. URL https://arxiv.org/abs/2502.14782.

[25] Kamaljyoti Nath, Khemraj Shukla, Victor C. Tsai, Umair bin Waheed, Christian Huber, Omer Alpak, Chuen-Song Chen, Ligang Lu, and Amik St-Cyr. Leveraging Deep Operator Networks (DeepONet) for Acoustic Full Waveform Inversion (FWI). *arXiv preprint arXiv:2504.10720*, 2025. URL https://arxiv.org/abs/2504.10720.

[26] Irina Higgins. Generalizing universal function approximators. *Nature Machine Intelligence*, 3(3):192–193, 2021.

[27] Ziming Liu, Yixuan Wang, Sachin Vaidya, Fabian Ruehle, James Halverson, Marin Soljačić, Thomas Y Hou, and Max Tegmark. Kan: Kolmogorov-arnold networks. *arXiv preprint arXiv:2404.19756*, 2024.

[28] Diab W. Abueidda, Panos Pantidis, and Mostafa E. Mobasher. DeepOKAN: Deep operator network based on Kolmogorov Arnold networks for mechanics problems. *Computer Methods in Applied Mechanics and Engineering*, 436:117699, 2025. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2024.117699. URL https://www.sciencedirect.com/science/article/pii/S0045782524009538.

[29] Khemraj Shukla, Juan Diego Toscano, Zhicheng Wang, Zongren Zou, and George Em Karniadakis. A comprehensive and FAIR comparison between MLP and KAN representations for differential equations and operator networks. *Computer Methods in Applied Mechanics and Engineering*, 431:117290, 2024. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2024.117290. URL https://www.sciencedirect.com/science/article/pii/S0045782524005462.

[30] Sanghyun Lee and Yeonjong Shin. On the Training and Generalization of Deep Operator Networks. *SIAM Journal on Scientific Computing*, 46(4):C273–C296, 2024. doi: 10.1137/23M1598751. URL https://doi.org/10.1137/23M1598751.

[31] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

[32] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[33] Ivo Babuška and Jens M Melenk. The partition of unity method. *International Journal for Numerical Methods in Engineering*, 40(4):727–758, 1997.

[34] Kookjin Lee, Nathaniel A. Trask, Ravi G. Patel, Mamikon A. Gulian, and Eric C. Cyr. Partition of unity networks: deep hp-approximation. *CoRR*, abs/2101.11256, 2021. URL https://arxiv.org/abs/2101.11256.

[35] Nathaniel Trask, Amelia Henriksen, Carianne Martinez, and Eric Cyr. Hierarchical partition of unity networks: fast multilevel training. In *Mathematical and Scientific Machine Learning*, pages 271–286. PMLR, 2022.

[36] Ramansh Sharma and Varun Shankar. Ensemble and Mixture-of-Experts DeepONets For Operator Learning. *arXiv preprint arXiv:2405.11907*, 2025.

[37] Levi D McClenny and Ulisses M Braga-Neto. Self-adaptive physics-informed neural networks. *Journal of Computational Physics*, 474:111722, 2023.

[38] Sokratis J Anagnostopoulos, Juan Diego Toscano, Nikolaos Stergiopulos, and George Em Karniadakis. Residual-based attention in physics-informed neural networks. *Computer Methods in Applied Mechanics and Engineering*, 421:116805, 2024.

[39] Somdatta Goswami, Aniruddha Bora, Yue Yu, and George Em Karniadakis. Physics-informed deep neural operator networks. In Timon Rabczuk and Klaus-Jürgen Bathe, editors, *Machine Learning in Modeling and Simulation: Methods and Applications*, pages 219–254. Springer International Publishing, Cham, 2023. ISBN 978-3-031-36644-4. doi: 10.1007/978-3-031-36644-4_6. URL https://doi.org/10.1007/978-3-031-36644-4_6.

[40] Katiana Kontolati, Somdatta Goswami, Michael D. Shields, and George Em Karniadakis. On the influence of over-parameterization in manifold based surrogates and deep neural operators. *Journal of Computational Physics*, 479:112008, April 2023. ISSN 0021-9991. doi: 10.1016/j.jcp.2023.112008. URL http://dx.doi.org/10.1016/j.jcp.2023.112008.

[41] Ahmad Peyvan, Vivek Oommen, Ameya D. Jagtap, and George Em Karniadakis. RiemannONets: Interpretable neural operators for Riemann problems. *Computer Methods in Applied Mechanics and Engineering*, 426:116996, 2024. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2024.116996. URL https://www.sciencedirect.com/science/article/pii/S0045782524002524.

[42] Hanxun Jin, Boyu Zhang, Qianying Cao, Enrui Zhang, Aniruddha Bora, Sridhar Krishnaswamy, George Em Karniadakis, and Horacio D. Espinosa. Characterization and Inverse Design of Stochastic Mechanical Metamaterials Using Neural Operators. *Advanced Materials*, page 2420063, 2025. ISSN 0935-9648, 1521-4095. doi: 10.1002/adma.202420063.

[43] George Karniadakis and Spencer J Sherwin. *Spectral/hp element methods for computational fluid dynamics*. Oxford University Press, 2005.

[44] Amin Paykani, Hamed Chehrmonavari, Athanasios Tsolakis, Terry Alger, William F. Northrop, and Rolf D. Reitz. Synthesis gas as a fuel for internal combustion engines in transportation. *Progress in Energy and Combustion Science*, 90:100995, 2022. ISSN 0360-1285. doi: https://doi.org/10.1016/j.pecs.2022.100995. URL https://www.sciencedirect.com/science/article/pii/S0360128522000041.

[45] David G Goodwin, Harry K Moffat, and Raymond L Speth. Cantera: An Object-oriented Software Toolkit for Chemical Kinetics, Thermodynamics, and Transport Processes. Version 2.3.0. *Zenodo*, 2017. doi: 10.5281/zenodo.170284.

[46] Kamaljyoti Nath, Varun Kumar, Daniel J Smith, and George Em Karniadakis. A Digital Twin for Diesel Engines: Operator-infused Physics-Informed Neural Networks with Transfer Learning for Engine Health Monitoring. *arXiv preprint arXiv:2412.11967*, 2025.

Appendices

- **Appendix A**: A brief discussion on the network architectures considered in the present study.
- **Appendix B**: Include additional discussion on the methodology considered.
  - **Appendix B.1**: Additional discussion on implementation of two-step training of DeepONet.
  - **Appendix B.2**: Discussion on calculation of adaptive loss function.
  - **Appendix B.3**: Discussion on implementation of mass-conserving DeepONet.
- **Appendix C**: Additional numerical results which are not included in the main content are presented
  - **Appendix C.1**: Additional results for syngas problem. In Appendix C.1.1, we have shown the convergence of all the DeepONet method for syngas problem.
  - **Appendix C.2**: Additional results for GRI-Mech 3.0 problem.
  - **Appendix C.3**: Implementation and additional result for AMORE in FNO.
- **Appendix D**: Additional discussion on computation cost.

A Network Architecture

In the present study, we have considered ResNet as one of the choices for the branch and trunk network in DON, and KAN as one of the choices for the trunk network and ResNet as choice of branch network in DOK. In this section, we briefly discuss the architecture of ResNet and KAN.

A.1 ResNet with MLP

As discussed earlier, we have considered ResNet (Residual network) with MLP for the branch and the trunk networks. ResNet consists of multiple residual blocks with skipped connections after every two layers. We have shown a schematic of a residual block in Fig. A.1. ResNet helps in training very deep networks, preventing overfitting, better gradient flow, and easier optimization. The residual block with input $\boldsymbol{x}$ can be written as

$$\boldsymbol{z}_1 = \sigma(\boldsymbol{x}\boldsymbol{W}_1 + \boldsymbol{b}_1) \tag{A.1a}$$

$$\boldsymbol{z}_2 = \boldsymbol{z}_1\boldsymbol{W}_2 + \boldsymbol{b}_2 \tag{A.1b}$$

$$\boldsymbol{z} = \sigma(\boldsymbol{z}_2 + \boldsymbol{x}) \tag{A.1c}$$

where $\boldsymbol{W}_1$ and $\boldsymbol{b}_1$ are the weights and biases of the first layer of the residual block and $\boldsymbol{W}_2$ and $\boldsymbol{b}_2$ are the weights and biases of the second layer of the residual block, σ is the activation function considered. We also like to mention that in the first residual block, there might be a mismatch in the dimension. Thus, the summation in Eq. (A.1c) may not be possible as in the present study. In this case, we consider a linear projection network (no activation function) with trainable weights and biases. In Table C.2, we have shown the number of parameters for the ResNet considered. The second term in the number of parameters refers to the parameters of the projection network.

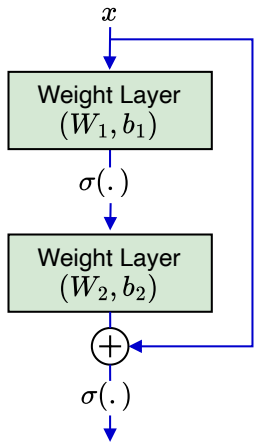


Fig. A.1: Schematic of a residual block of ResNet showing the residual connection.

A.2 Kolmogorov-Arnold Networks: KAN

Based on the Kolmogorov Arnold theorem, Liu et al. [27] proposed Kolmogorov Arnold networks (KAN). The basic idea of KAN is that a multivariate continuous function can be written as a sum of the finite composition of continuous functions of the single variables. In the present study, we have considered the Jacobi polynomials as the choice of polynomial function. Below, we have shown the implementation of KAN with the Jacobi polynomials in pseudo-Python code

```
# Python implementatiaon
A = t   # time after normalization, size: (n_t, 1)
for i in range(L):                      # L: number of Layers
    A = sin(A)
    poly = Polynomial(A)                    # Output size: (n_t, in_dim, order +1)
    A = einsum("ijk,jlk->il", poly, coeff[i])
```

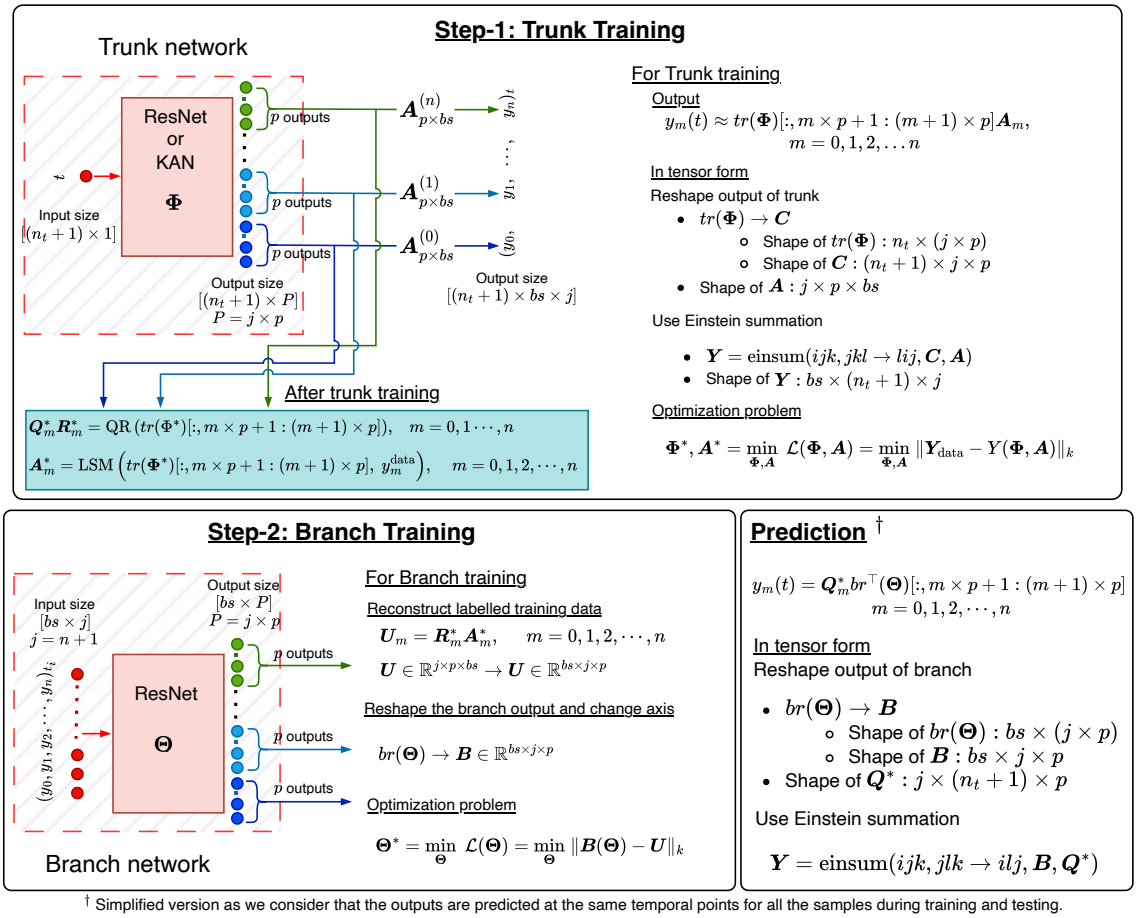
where $\mathrm{t} \in \mathbb{R}^{n_t \times 1}$ is the input time points, L is the number of layers in KAN, and "Polynomial" is the function that generates the polynomial of A of desired order. The "coeff[i]" are the trainable parameters of the i^{th} layer with size in_dim $\times$ out_dim $\times$ order+1. The size of "poly" is $n_t \times$ in_dim $\times$ order+1. We consider the polynomials defined by a recursive relationship discussed by Karniadakis and Sherwin [43] with α and β values of the polynomial as discussed in Table C.2. We would also like to discuss the importance of the sin( ) function. We know that the domain of a Jacobi polynomial is $[-1,1]$ while the range is $\mathbb{R}$. Thus, after the first layer, the input to the Jacobi function may not be in $[-1,1]$; therefore, to keep the domain within the range $[-1,1]$, we consider the sin( ) function. It will ensure that the domain of the Jacobi function is always $[-1,1]$.

B Additional discussion on Methodology

We have discussed AMORE, implementation of DeepONet and self-adaptive loss function in the main text. Here, we present additional discussion which are not included in the main text.

B.1 Two-step training of DeepONet

We have discussed the implementation of two-step training in Section 3.4. A schematic diagram for the two-step training of DeepONet is shown in Fig. B.1.



† Simplified version as we consider that the outputs are predicted at the same temporal points for all the samples during training and testing.

Fig. B.1: **Schematic of two-step training of DeepONet for the proposed DeepONet** showing branch and trunk networks and multiple outputs along with training detail. We have shown the trunk network in the top left in the dashed red box with input $t\in\mathbb{R}^{(n_t+1)\times 1}$ and output $tr(\mathbf{\Phi})\in\mathbb{R}^{(n_t+1)\times P}$. Where (n_t+1) is the number of time points where the output needs to be predicted, and $P=j\times p$ is the number of neurons in the output layer. p is the number of basis functions for each state variable. We have considered ResNet or KAN (with trainable parameters $\mathbf{\Phi}$) as the choice of network architecture for the trunk network. The output is predicted by multiplying the trunk output with a trainable tensor, as shown in Eq. (18). Thus, in the first step, the optimization process is to obtain the optimal values of the parameters $\mathbf{\Phi}$ and $\boldsymbol{A}$ by minimizing a loss function between the labeled data and predicted output. The output and optimization problem for trunk training is shown in the top right. After the trunk training, the next step is the QR decomposition of the trunk output and recalculation of $\boldsymbol{A}$ using the least square approximation. The branch network is shown in the bottom right in the dashed red box with input $\boldsymbol{Y}_0\in\mathbb{R}^{bs\times j}$, and output $br(\mathbf{\Theta})\in\mathbb{R}^{bs\times P}$. Where bs is the number of samples and $P=j\times p$ is the number of neurons in the output layer. p is the number of coefficients for each state variable. We consider ResNet with trainable parameters $\mathbf{\Theta}$ as the choice of network architecture for the branch network. The parameters of the branch ($\mathbf{\Theta}$) are optimized using a data loss with respect to a reconstructed labelled data $\boldsymbol{U}$ from the trunk output. The optimization process is shown in the bottom-middle. The prediction using the re-parameterized DeepONet is shown in the bottom-right and discussed in Section 3.4.3. We also consider adaptive weight in the loss function for both training of trunk and branch networks. These are discussed in Sections 3.4.1 and 3.4.2 respectively.

We consider three loss functions: the MSE loss function and the two adaptive loss functions. The adaptive loss functions for the trunk training are straightforward and similar to the single-step DeepONet and discussed in Section 3.4.2. The implementation of the adaptive loss function for the branch network required marginal modification. The loss function for branch training is discussed below:

♦ **Non-adaptive MSE loss function:** This loss function is defined as the mean square error between the branch output ($\boldsymbol{B}$) and the reconstructed labeled data ($\boldsymbol{U}$) and is similar to the non-adaptive loss function discussed earlier.
♦ **Adaptive loss function Type-A:** We define this loss function as data loss with adaptive weights $\mathcal{W}_a$ associated with state variable a

$$\text{(B.1)}\quad \mathcal{L}(\boldsymbol{\Theta}) = \sum_{a=0}^{j-1} \mathcal{W}_a \left[\frac{1}{bs \times p} \sum_{b=1}^{bs} \sum_{c=1}^{p} (B_{bac} - U_{bac})^2 \right], \qquad \begin{array}{l} j \rightarrow \text{No. of states variable,} \\ p \rightarrow \text{No. of basis functions for each state variable,} \\ bs \rightarrow \text{No. of samples} \end{array}$$

♦ **Adaptive loss function Type-B:** We define this loss function as data loss with adaptive weights $\mathcal{W}_{ba}$ associated with state variable a and sample b.

$$\text{(B.2)}\quad \mathcal{L}(\boldsymbol{\Theta}) = \sum_{a=0}^{j-1} \sum_{b=1}^{bs} \mathcal{W}_{ba} \left[\frac{1}{p} \sum_{c=1}^{p} (B_{bac} - U_{bac})^2 \right], \qquad \begin{array}{l} j \rightarrow \text{No. of states variable,} \\ p \rightarrow \text{No. of basis functions for each state variable,} \\ bs \rightarrow \text{No. of samples} \end{array}$$

In these cases, as well, the adaptive weights, similar to the weights discussed in Sections 3.3.1 and 3.3.2, the $\mathcal{W}_a$ and $\mathcal{W}_{ab}$ are updated using a gradient-free update scheme. However, since $\boldsymbol{U}$ is not true labeled data, we calculate the relative L_2 error from the final output of DeepONet, in the physical domain, and the true labeled data ($\boldsymbol{Y}$). The prediction of the final DeepONet output is discussed in the next section (Section 3.4.3)

B.2 Calculation for adaptive loss function

We have proposed two adaptive loss functions in Section 3.3, called Type-A (in Section 3.3.1) and Type-B (in Section 3.3.2). Also, we have considered adaptive loss functions in the two-step training of DeepOnet as discussed in Section 3.4 and Appendix B.1. In this section, we briefly discussed the calculation of the adaptive loss function.

The calculation of the adaptive weights for the loss functions is as follows: the size of the true output and the predicted outputs are $bs \times (n_t + 1) \times j$, where bs is the number of samples, $(n_t + 1)$ is the number of time points considered (trunk input size), and j is the number of state variables. In the case of adaptive loss of Type-A, the X_a is calculated as

$$\text{(B.3)}\quad X_j = \text{mean}\left(\frac{L_2 \text{ norm}\left(\boldsymbol{Y}_{bs \times n_t \times j} - \hat{\boldsymbol{Y}}_{bs \times n_t \times j},\ \text{axis} = n_t \right)}{L_2 \text{ norm}\left(\boldsymbol{Y}_{bs \times n_t \times j},\ \text{axis} = n_t \right)},\quad \text{axis} = bs \right)$$

Similarly, the X_{ba} in the case of adaptive loss function Type-B is calculated as

$$\text{(B.4)}\quad X_{bs,j} = \frac{L_2 \text{ norm}\left(\boldsymbol{Y}_{bs \times n_t \times j} - \hat{\boldsymbol{Y}}_{bs \times n_t \times j},\ \text{axis} = n_t \right)}{L_2 \text{ norm}\left(\boldsymbol{Y}_{bs \times n_t \times j},\ \text{axis} = n_t \right)}$$

As discussed in Section 3.3, the sum of the adaptive weights is constant. In the case of Type-A, adaptive loss function, the sum of all the weights $\mathcal{W}_a$ at any epoch is always $\mathcal{R}_1$ as shown in Eq. (B.5).

$$\text{(B.5)}\quad \sum_{a=0}^{j-1} \mathcal{W}_a = \sum_{a=0}^{j-1} \frac{X_a}{\sum_{d=0}^{j-1} X_d} \times \mathcal{R}_1 = \frac{\sum_{a=0}^{j-1} X_a}{\sum_{d=0}^{j-1} X_d} \times \mathcal{R}_1 = \mathcal{R}_1$$

Similarly, we can show that in the case of Type-B adaptive loss function also the summation of all the adaptive weights $\mathcal{W}_{ab}$ is $\mathcal{R}_2$ at any epoch.

B.3 Discussion on implementation of mass-conserving DeepONet

In Section 3.5, we have discussed the basic philosophy of the mass-conserving DeepONet. In this section, we discuss the implementation of the mass-conserving DeepONet. The method involved mapping the mass fraction to a lower dimension, training the DeepONet in the lower dimension, and converting back to the physical domain using an inverse mapping.

Our approach to constructing such maps is inspired by mapping widely used in spectral element simulation of PDES [43], where a triangle is mapped to a square, i.e., a collapsed two-dimensional coordinate system is mapped to a square. We extend such mappings to n-dimensional coordinates. In particular, the forward mapping (ref. Eq. (28))

$$\text{(B.6)}\quad z_1, z_2, \cdots z_{n-1} = g(y_1, y_2, \cdots y_n),$$

is given by:

$$z_k = \frac{y_k}{1 - \sum_{j=1, j\neq k}^{n-1} y_j}, \quad 1 \leq k \leq n-2 \tag{B.7a}$$

$$z_{n-1} = y_{n-1} \tag{B.7b}$$

To find the inverse map ($g^{-1}(\cdot)$) (ref. Eq. (29))

$$y_1, y_2, \cdots y_n = g^{-1}(z_1, z_2, \cdots z_{n-1}), \tag{B.8}$$

we start from the forward map

$$z_k = \frac{y_k}{d_k}, \quad k = 1, \ldots, n-2, \quad \text{where} \quad d_k = 1 - \sum_{j=1, j\neq k}^{n-1} y_j, \tag{B.9}$$

By substituting $y_k = z_k d_k$, we obtain:

$$d_k = 1 - \sum_{j=1, j\neq k}^{n-1} z_j d_j. \tag{B.10}$$

This results in a system of linear equations:

$$d_k + \sum_{j\neq k}^{d} z_j d_j = 1, \quad \text{for } k = 1, \ldots, n-2. \tag{B.11}$$

This system can be written in matrix form as:

$$\boldsymbol{A}\boldsymbol{d} = \boldsymbol{b}, \tag{B.12}$$

where: $\boldsymbol{d} = [d_1, d_2, \ldots, d_{n-2}]^T$ is the vector of unknowns, $\boldsymbol{b} = [1, 1, \ldots, 1]^T$ is a vector of ones, and $\boldsymbol{A}$ (not same as considered in two-step training) is a $(n-2) \times (n-2)$ coefficient matrix defined as:

$$\boldsymbol{A}_{kj} = \begin{cases} 1, & \text{if } k = j, \\ z_j, & \text{if } k \neq j. \end{cases} \tag{B.13}$$

Once the above linear system is solved, the mass fraction vector can be computed via:

$$y_k = z_k d_k, \quad 1 \leq k \leq n-2 \tag{B.14a}$$

$$y_{n-1} = z_{n-1} \tag{B.14b}$$

$$y_n = 1 - \sum_{k=1}^{n-1} y_k. \tag{B.14c}$$

To implement the above transformation in the DeepONet, the first step is to convert the mass fraction using the forward map, Eq. (B.6). The next step is to formulate the DeepONet learning problem, which can be written as

$$\boldsymbol{Z}(t) \approx \mathcal{G}_\theta : \boldsymbol{Z}_0 \mapsto \boldsymbol{Z}(\boldsymbol{Z}_0)(t) \tag{B.15}$$

where $\boldsymbol{Z}(t) = [T(t), z_1(t), z_2(t), \cdots z_{n-1}(t)]^\top$ and $\boldsymbol{Z}_0 = [T, z_1, z_2, \cdots z_{n-1}]_0^\top$. Once the DeepONet is trained, we convert the predicted output to the physical domain using the inverse mapping, Eq. (B.8). We also want to mention that in addition to the forward and inverse mapping, we also consider normalization of the input and output in the transformed domain. These are discussed in computational examples.

C Additional Results

C.1 Additional result for the syngas problem

We have discussed the results for the syngas problem in Section 4. In this section, we will present additional results for the syngas problem not included in Section 4. In Table C.1, we have shown the minimum and maximum values of the training and testing dataset of the state variables of the syngas problem. In Table C.2, we have shown the network detail for DON, and DOK, including activation function, layers, number of parameters, etc. In Fig. 5 (in Section 4), we have shown the violin plots for relative L_2 error for four methods considered. Here, in Fig. C.1, we have shown the violin plots for all nine methods considered. The following additional results are presented in this section:

Additional tables and figures for results of the syngas problem.

Table/Figure	Brief description
Fig. C.1	Syngas problem, violin plot of relative L_2 error.
Fig. C.2	Syngas problem, additional sample test result when consider 2S-Ad-B
Fig. C.3	Syngas problem, extrapolation results
Table C.3	Syngas problem, Mass conserving DeepONet relative L_2 error
Fig. C.4	Syngas problem, Recursive prediction, Violin plot of relative L_2 error
Fig. C.5	Syngas problem, Recursive prediction, sample results
Appendix C.1.1:	Convergence of relative L_2 error for different methods considered
Fig. C.6	Convergence of relative L_2 error for one-step training.
Fig. C.7	Convergence of relative L_2 error for trunk training using two-step training.
Fig. C.8	Convergence of relative L_2 error for branch training using two-step training.
Fig. C.9	Convergence of relative L_2 error for trunk and branch training in two-step training of mass-conserving DeepOKAN.

Table C.1: **Syngas problem, minimum and maximum value** of the training and testing dataset for the syngas problem for the state variables (temperature and mass fraction for each of the 11 species).

State variable	Training dataset		Testing dataset	
	Minimum Value	Maximum Value	Minimum Value	Maximum Value
Temperature	1.000e+03	2.855e+03	1.000e+03	2.852e+03
H_2	1.060e-04	4.938e-03	1.060e-04	4.938e-03
O_2	1.977e-02	2.033e-01	1.985e-02	2.033e-01
O	2.403e-12	2.492e-02	2.454e-12	2.492e-02
OH	1.429e-14	1.131e-02	1.498e-14	1.129e-02
H_2O	3.315e-16	3.445e-02	3.580e-16	3.443e-02
H	8.988e-15	1.663e-03	9.206e-15	1.634e-03
HO_2	2.707e-13	1.134e-04	2.765e-13	1.134e-04
CO	4.326e-02	3.431e-01	4.356e-02	3.431e-01
CO_2	6.652e-12	2.974e-01	6.794e-12	2.974e-01
HCO	1.820e-17	2.406e-06	1.936e-17	2.405e-06
N_2	4.840e-01	5.860e-01	4.840e-01	5.860e-01

Table C.2: **DeepONet network details considered for the Syngaas problem.** We consider two different types of architectures. In DON (DeepONet), ResNet is considered in both the branch and trunk, while in DOK (DeepOKAN), ResNet is considered in the branch, and KAN is considered in the trunk. The ResNet size $[12-200\times10-1140]$ indicates that there are 12 neurons in the input layer, 10 hidden layers with 200 neurons in each layer, and the output layer contains 1140 neurons. The KAN $[1-75\times4-1140]$ indicates 4 layers with 75 edges. We consider Jacobi polynomials for KAN. We discussed the ResNet and KAN architecture in Appendix A. The output 1140 neurons in the output layer are divided into 12 parts, each part with 95 neurons for each state variable, i.e., $p = 95$. In the "Number of parameters", the second number, in the case of ResNet, indicates the number of parameters in the projection network required in the first layer due to size mismatch. The "sin( )" in KAN is an additional function and is discussed in Appendix A.

		DON	DOK
Branch	Network Type	ResNet	ResNet
	Network size	$[12-200\times10-1140]$	$[12-200\times10-1140]$
	Activation function	tanh( )	tanh( )
	Number of parameters	$593540+2600$	$593540+2600$
Trunk	Network Type	ResNet	KAN 3rd order Jacobi ($\alpha = 1.0$, $\beta = 1.0$)
	Network size	$[1-200\times10-1140]$	$[1-75\times4-1140]$
	Activation function	tanh( )	sin( ) (*ref Appendix A)
	Number of parameters	$591340+400$	409800
Total parameters		1187880 (1.2 M)	1005940 (1.0 M)

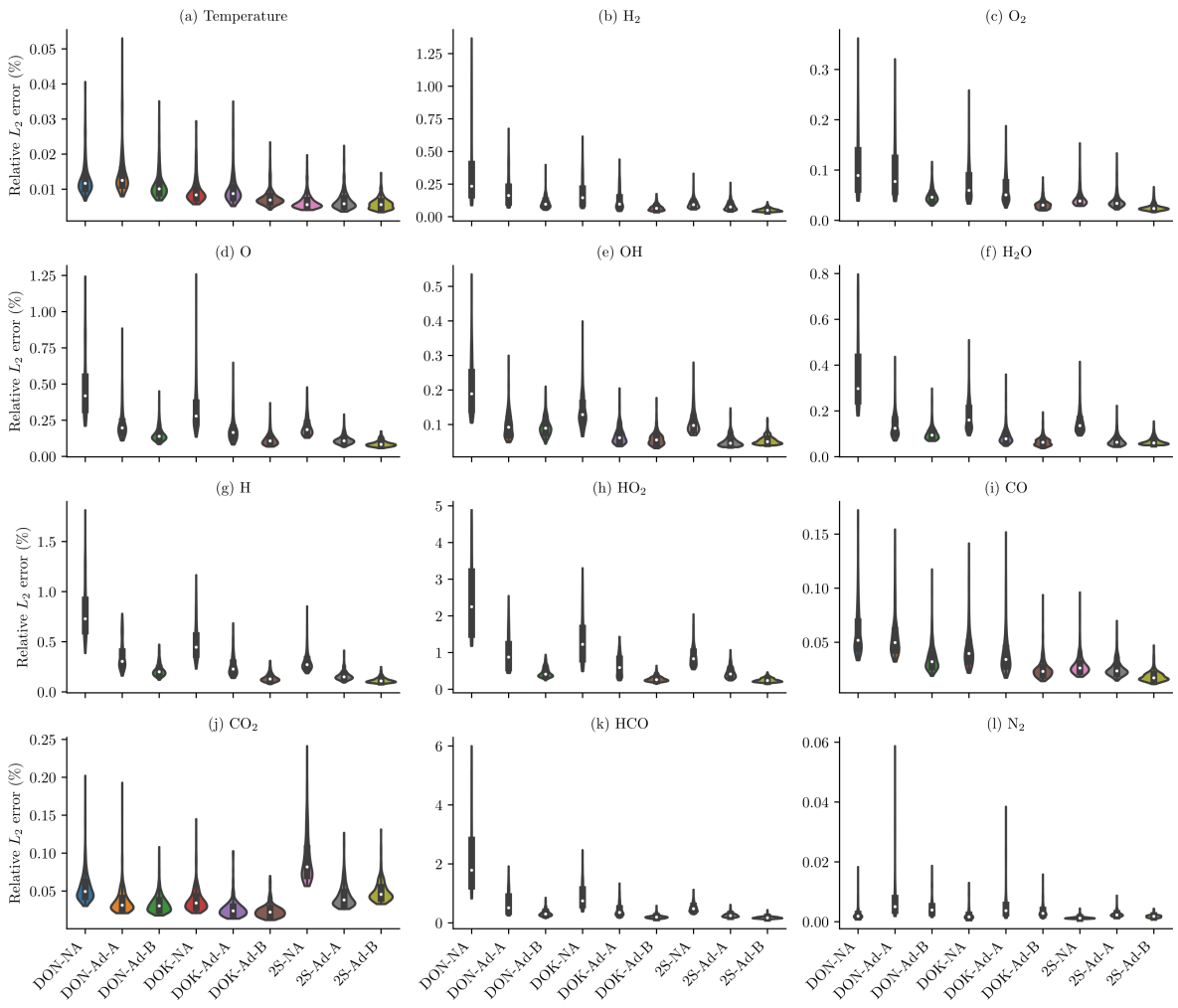
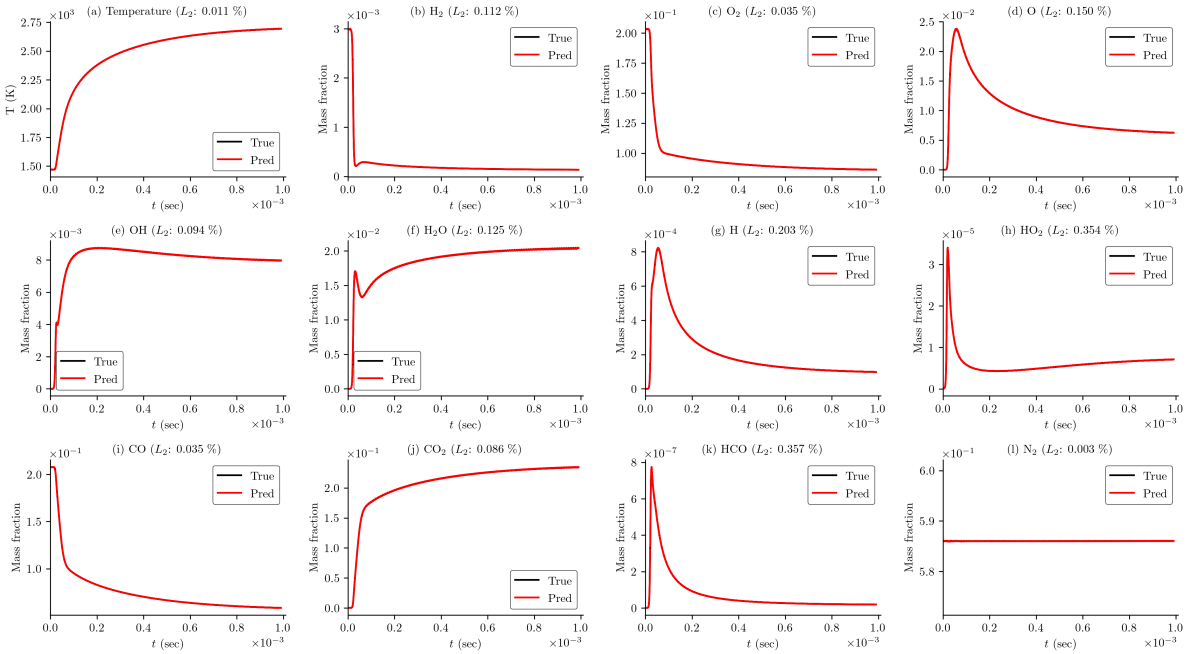
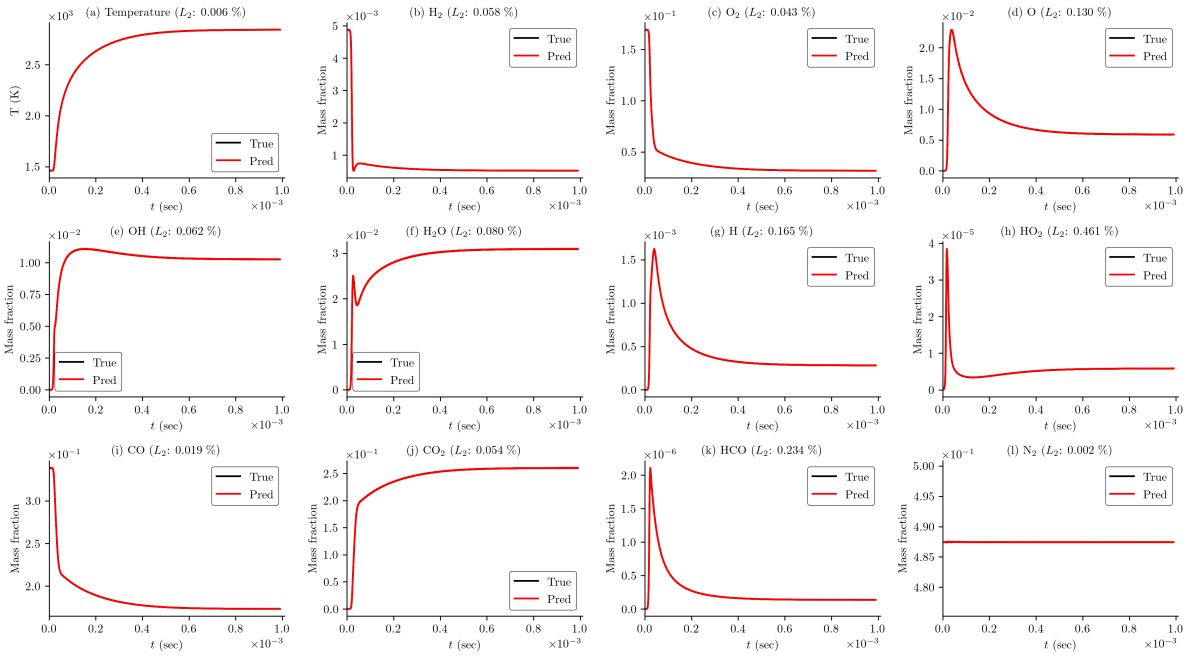


Fig. C.1: **Syngas problem, Violin plot** of the relative L_2 error of the reconstructed prediction of the state variables of the test dataset. We observed that the predicted results using DON-NA have the highest mean of the relative L_2 error compared to the other methods. Furthermore, DON-NA also has a higher standard deviation of the relative L_2 error. The mean and standard deviation are reduced as adaptive loss functions are considered. We would particularly like to mention that the state variables corresponding to the dynamics of HCO and HO2 shown in (k) and (h), respectively, have higher errors, which are reduced when we consider the adaptive loss function. We have observed that the DON-Ad-B and 2S-Ad-B have similar accuracy; however, 2S-Ad-B has smaller standard deviations.



(a) Test Sample 2 (2S-Ad-B)

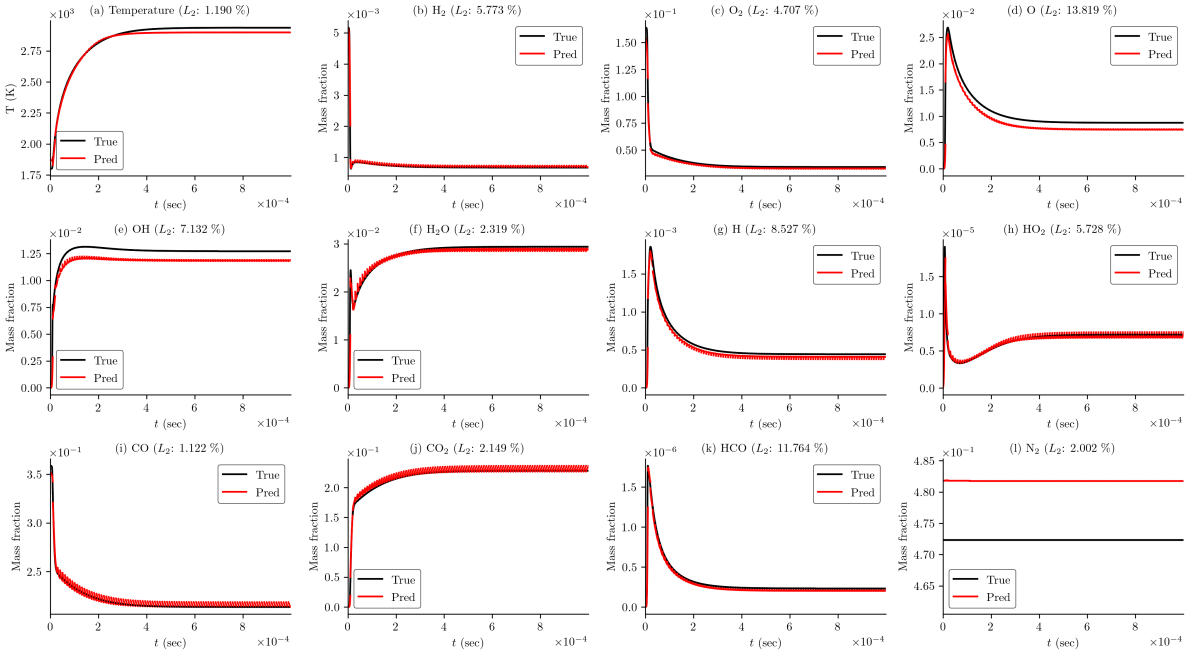


(b) Test Sample 3 (2S-Ad-B)

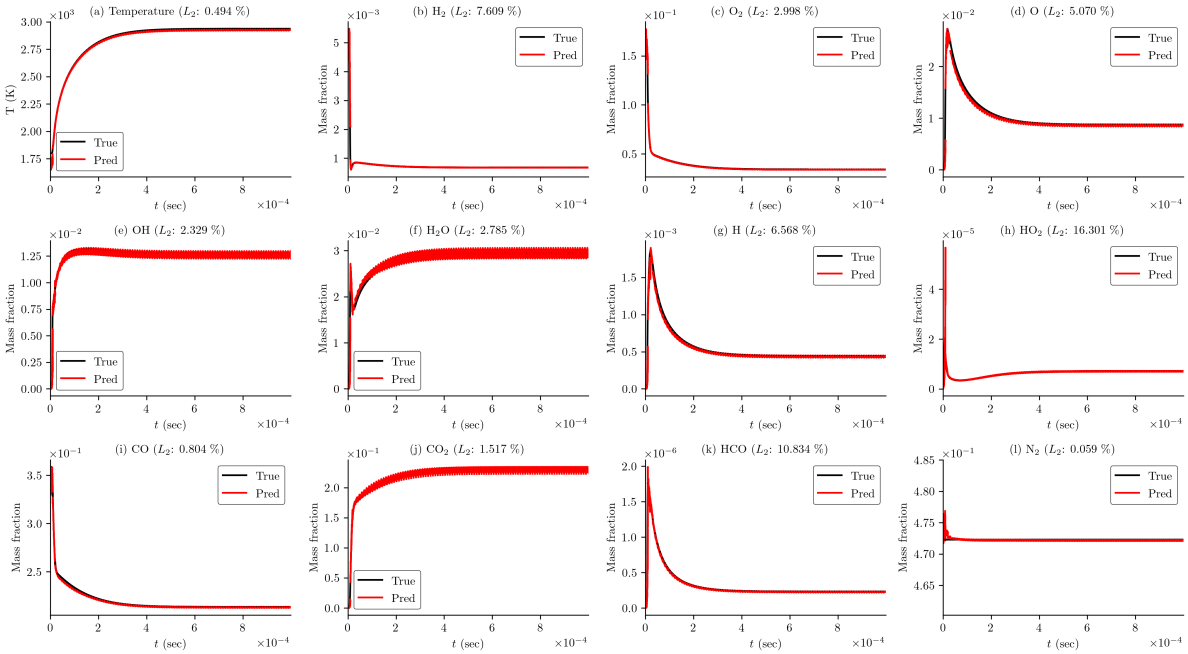
Fig. C.2: **Syngas problem, additional sample test result, 2S-Ad-B:** Plot showing two representative sample results from the test dataset when predicted 2S-Ad-B, i.e., DeepOKAN trained using the two-step training method and adaptive loss function Type-B. The number in the bracket indicates the relative L_2 errors for each species (calculated as discussed in Section 4.3). The sample result corresponds to one of the higher errors in prediction. The predicted sample shows good accuracy with the true value.

Table C.3: **Syngas problem, Mass conserving DeepONet:** Relative L_2 error in training and testing for mass conserving DeepONet (Section 3.5) for syngas problem for three independent runs. For comparison, we have shown the training and testing error correspond to DeepONet trained without considering mass conservation, i.e., 2S-Ad-B. The convergence of the relative L_2 error in trunk and branch training for 2S-Ad-B-CoM is shown in Fig. C.9 (in Appendix C).

Case	Relative L_2 error (%)	
	Training	Testing
2S-Ad-B	0.041, 0.040, 0.041	0.041, 0.040, 0.041
2S-Ad-B-CoM	0.035, 0.037, 0.034	0.035, 0.037, 0.034



(a) DOK-Ad-B, Extrapolation sample # 2



(b) 2S-Ad-B, Extrapolation sample # 2

Fig. C.3: **Syngas problem, DeepONet extrapolation result # 2.** Plot showing the dynamics of the state variables of the syngas problem for extrapolation dataset #2 when predicted using (a) DOK-Ad-B and (b) 2S-Ad-B method. The predicted dynamics show good accuracy with the true dynamics.

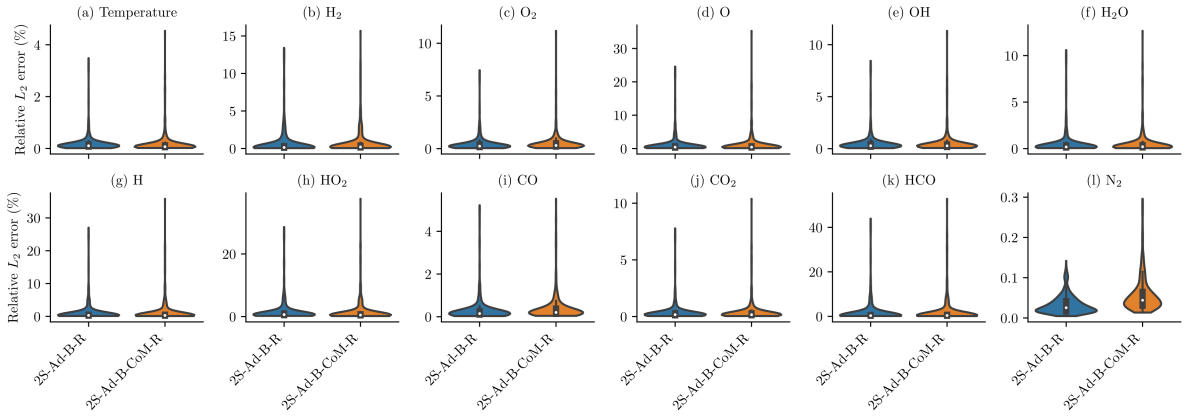
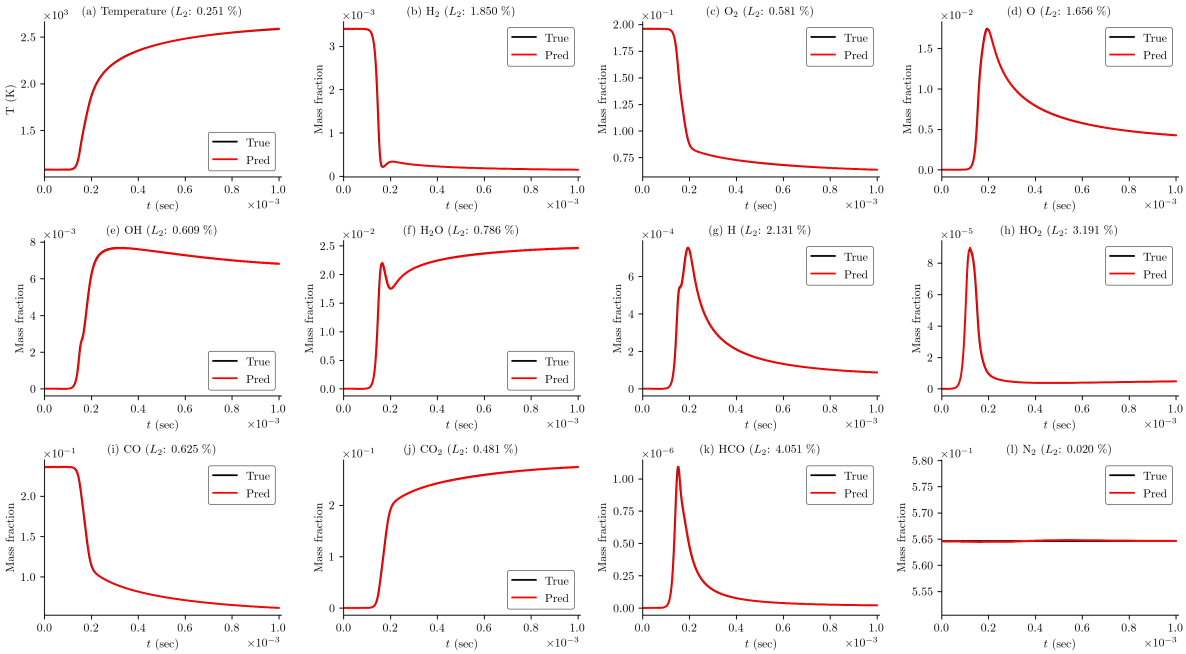
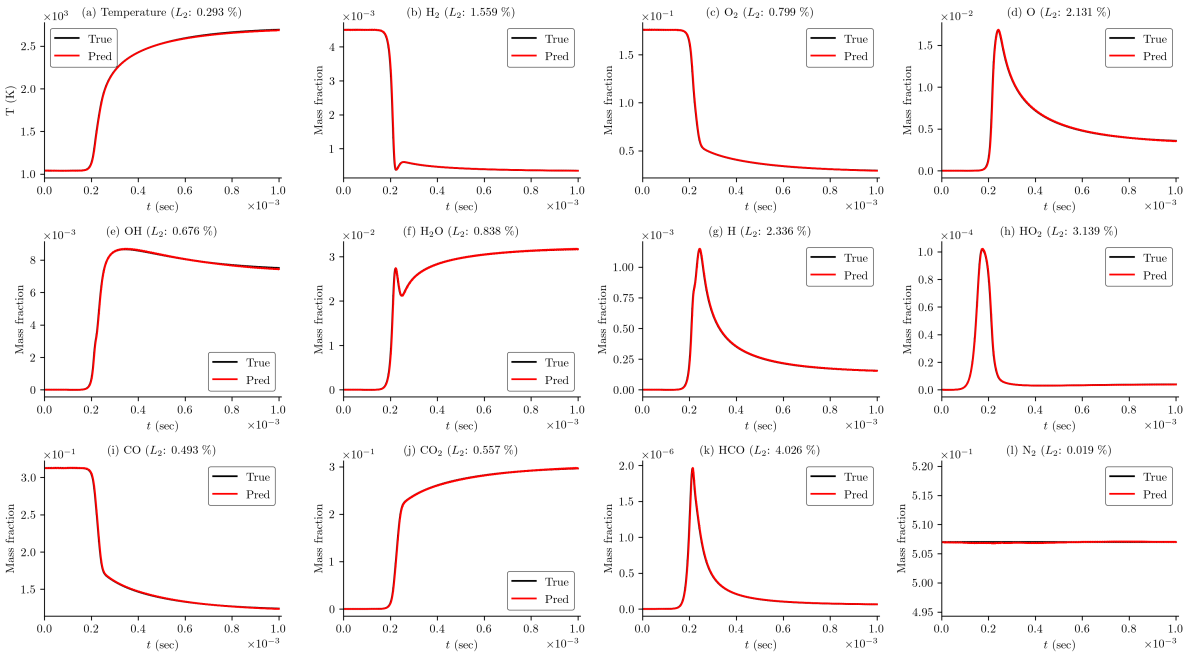


Fig. C.4: **Syngas problem, Recursive prediction, Violin plot** for relative L_2 error of the reconstructed prediction of the state variables of the test dataset. The outputs are predicted recursively as discussed in Section 3.6. The relative L_2 errors are calculated similarly as discussed in Section 4.3.



(a) Recursive prediction, Sample results 1



(b) Recursive prediction, Sample results 2

Fig. C.5: **Syngas problem, Recursive prediction, sample results.** Plot showing the dynamics of the state variables of the syngas problem for test dataset when predicted using recursive prediction. (a) shows the dynamics of the state variables corresponding to 90 percentile error in predicted state variable H_2O (0.786%), (b) shows the dynamics of the state variables corresponding to 90 percentile error in predicted state variable Temperature (0.29%). The plots show the dynamics of the predicted state variable when predicted recursively using the trained model 2S-Ad-B, which we called 2S-Ad-B-R. Both the plots show good accuracy in prediction.

C.1.1 Error Convergence for Syngas problem

In Fig. C.6, we have shown the convergence of relative L_2 error (calculated as discussed in Section 3.7) in training and testing for DON and DOK when trained using different loss functions. We have also shown the learning rate considered in the training. Similarly, the convergence of relative L_2 error in training for trunk training in the case of two-step training is shown in Fig. C.7 and that of training and testing in branch training in the case of two-step training is shown in Fig. C.8. These convergence plots show the convergence of relative L_2 error (calculated as discussed in Section 3.7) for three independent run for each method. We observed that all the three runs follow

similar converge path. Furthermore, the error in the case of adaptive loss functions is smaller than the non adaptive case. However, the non adaptive loss converge marginally faster than the adaptive loss functions. In Fig. C.9, we have shown the converge of relative L_2 error in the case of two-step method with mass conservation discussed in Section 4.4.

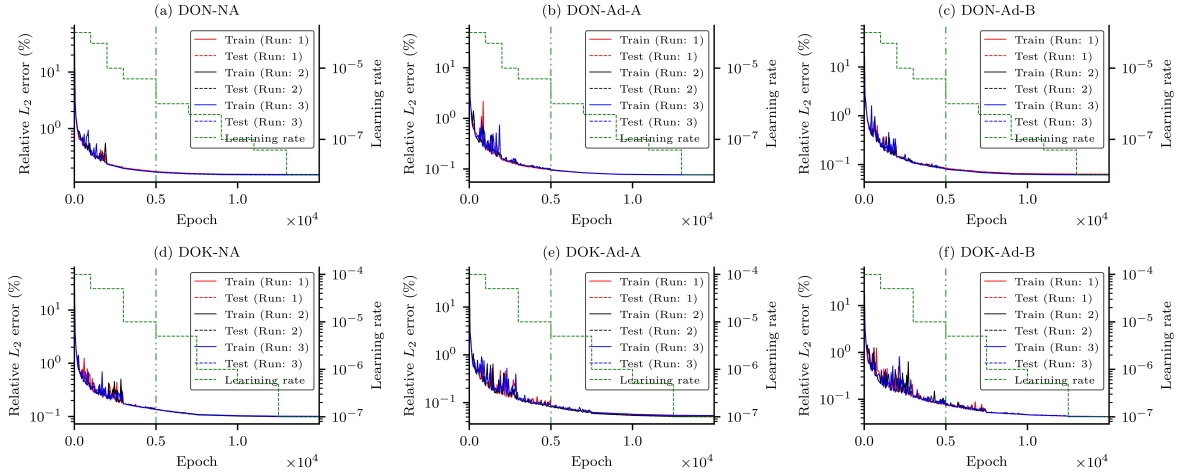


Fig. C.6: **Convergence of error for Syngas problem for DeepONet.** Convergence of mean of relative L_2 error in training and testing with epoch for different methods of DeepONet and loss functions considered. The plots show the convergence for the three independent runs. The three runs follow a similar convergence path with epochs. We have also shown the learning rate considered for each method. We have considered 60 mini-batch up to 5 k epoch and 120 mini-batch after 5 k epoch and indicated by the vertical line at 5 k epoch.

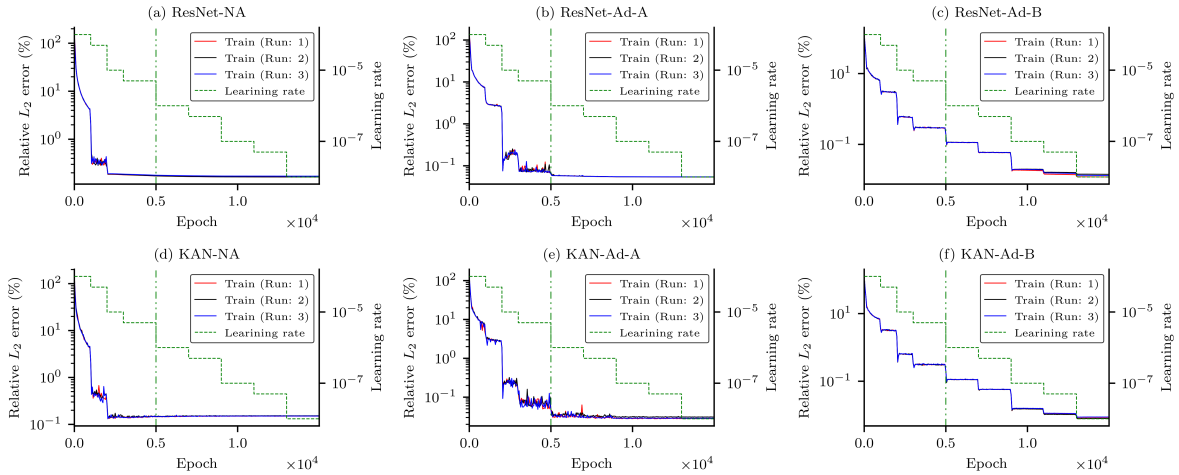


Fig. C.7: **Convergence of error for Syngas problem for trunk training when trained using two-step training of DeepONet.** Convergence of mean of relative L_2 error in trunk training with epoch for different methods of DeepONet and loss functions considered when trained using two-step training. The plots show the convergence for the three independent runs. The three runs follow a similar convergence path with epochs. We have also shown the learning rate considered for each method. We have considered 60 mini-batch up to 5 k epoch and 120 mini-batch after 5 k epoch and indicated by the vertical line at 5 k epoch.

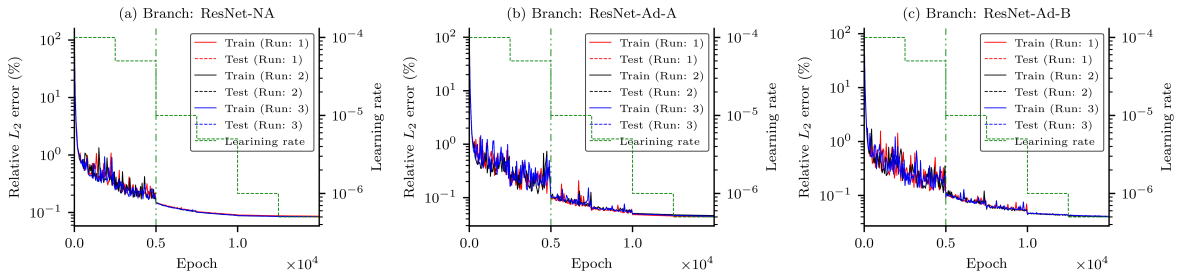


Fig. C.8: **Convergence of error for Syngas problem for branch training when trained using two-step training of DeepONet.** Convergence of mean of relative L_2 error in branch training for training and testing dataset with epoch for different methods of DeepONet and loss functions considered when trained using two-step training. The plots show the convergence for the three independent runs. The three runs follow a similar convergence path with epochs. We have also shown the learning rate considered for each method. We have considered 60 mini-batches up to 5 k epochs and 120 mini-batches after 5 k epochs, and indicated by the vertical line at 5 k epochs.

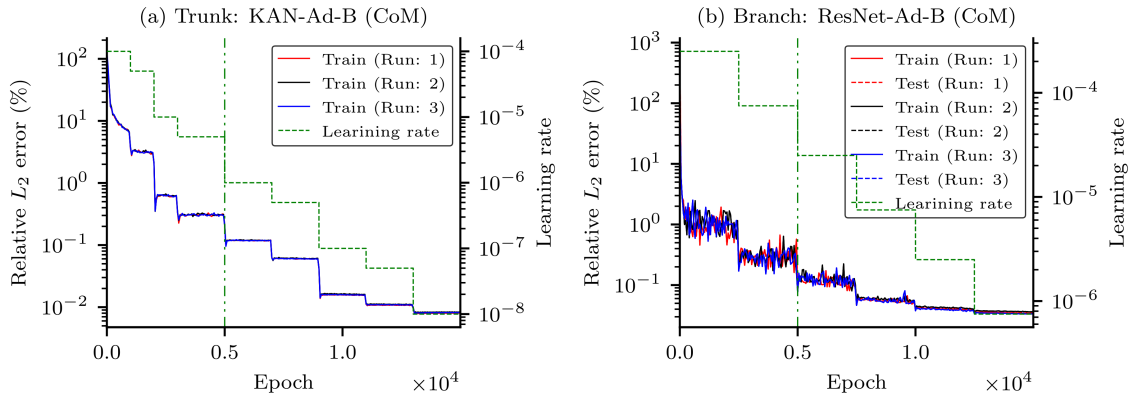


Fig. C.9: **Syngas problem, convergence Mass conserving DeepOKAN.** Convergence of error for Syngas problem for trunk training and branch training when trained using two-step training of DeepOKAN with mass conservation (ref Section 3.5). Convergence of mean of relative L_2 error in trunk and branch training with epoch for adaptive loss Type-B. The plots show the convergence for the three independent runs. The three runs follow a similar convergence path with epochs. We have also shown the learning rate considered for each method. We have considered 60 mini-batch up to 5 k epoch and 120 mini-batch after 5 k epoch and indicated by the vertical line at 5 k epoch.

C.2 GRI-Mech 3.0 problem

We have discussed the results for the GRI-Mech 3.0 problem in Section 5. In this section, we will present additional results for the GRI-Mech 3.0 problem not included in Section 5. In Table C.4, we have shown the minimum and maximum values of the training and testing dataset of the state variables of the GRI-Mech 3.0 problem. In Table C.5, we have shown the network detail for DOK, including activation function, layers, number of parameters, etc. In Fig. C.10, we show the convergence of relative L_2 error for trunk and branch training using two-step training of DeepOKAN. Predicted samples for the GRI Mech 3.0 problem show good agreement with the true solution as given in Fig. C.11.

Table C.4: **GRI Mech 3.0 problem, Minimum and maximum value** of the training and testing dataset for the GRI problem for the state variables (temperature and each of the 23 species).

State variable	Training dataset		Testing dataset	
	Minimum Value	Maximum Value	Minimum Value	Maximum Value
Temperature	1.000e+03	2.870e+03	1.000e+03	2.867e+03
H_2	7.214e-05	1.787e-02	7.763e-05	1.787e-02
H	5.110e-14	3.873e-03	5.961e-14	3.864e-03
O	1.160e-12	2.356e-02	1.334e-12	2.355e-02
O_2	7.253e-03	2.298e-01	7.791e-03	2.298e-01
OH	3.626e-14	1.921e-02	4.513e-14	1.921e-02
H_2O	1.104e-14	1.248e-01	1.614e-14	1.241e-01
HO_2	1.681e-12	1.428e-04	1.943e-12	1.428e-04
H_2O_2	1.595e-17	9.155e-06	2.187e-17	8.794e-06
CO	5.820e-03	2.482e-01	6.318e-03	2.482e-01
CO_2	3.257e-12	1.867e-01	3.767e-12	1.856e-01
HCO	5.465e-17	2.748e-06	7.557e-17	2.716e-06
CH_2O	9.630e-21	4.200e-08	1.439e-20	4.112e-08
N	2.698e-27	1.705e-06	2.975e-27	1.665e-06
NH	1.500e-28	2.212e-07	1.889e-28	2.133e-07
NH_2	3.043e-31	4.038e-08	4.532e-31	3.959e-08
NH_2	5.108e-33	2.286e-08	9.024e-33	2.260e-08
NNH	5.047e-19	1.079e-07	5.642e-19	1.074e-07
NO	7.873e-27	8.422e-03	8.644e-27	8.260e-03
NO_2	5.099e-37	3.982e-06	5.130e-37	3.881e-06
HNO	2.777e-30	5.054e-07	3.333e-30	5.049e-07
HNCO	2.370e-31	2.262e-08	3.628e-31	2.234e-08
NCO	5.533e-31	5.498e-09	7.369e-31	5.351e-09
N_2	5.421e-01	6.623e-01	5.422e-01	6.623e-01

Table C.5: **DeepONet size considered for GRI problem**. We consider two different types of architectures. In DON (DeepONet), ResNet is considered in both the branch and trunk, while in DOK (DeepOKAN), ResNet is considered in the branch, and KAN is considered in the trunk. The ResNet size $[12-200\times10-2280]$ indicates that there are 12 neurons in the input layer, 10 hidden layers with 200 neurons in each layer, and the output layer contains 2280 neurons. The KAN $[1-95\times5-2280]$ indicates five layers with 95 edges. We consider the Jacobi polynomial for KAN. We discussed the ResNet and KAN architecture in Appendix A. The output 2280 neurons are divided into 24 parts, each part with 95 neurons for each state variable, i.e., $p = 95$. In the "Number of parameters", the second number, in the case of ResNet's, indicates the number of parameters in the projection network required in the first layer due to size mismatch. The "sin()" in KAN is an additional function and is discussed in Appendix A.

		DOK
Branch	Network Type	ResNet
	Network size	$[24-250\times10-2280]$
	Activation function	tanh( )
	Number of parameters	1143280 + 6250
Trunk	Network Type	KAN 3rd order Jacobi ($\alpha = 1.0, \beta = 1.0$)
	Network size	$[1-95\times5-2280]$
	Activation function	sin( ) (*ref Appendix A)
	Number of parameters	1011180
Total parameters		2160710 (2.2 M)

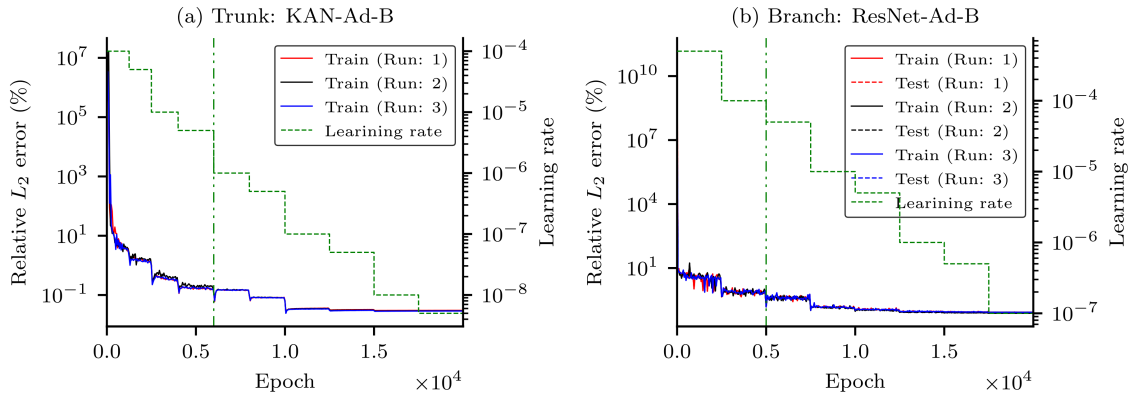
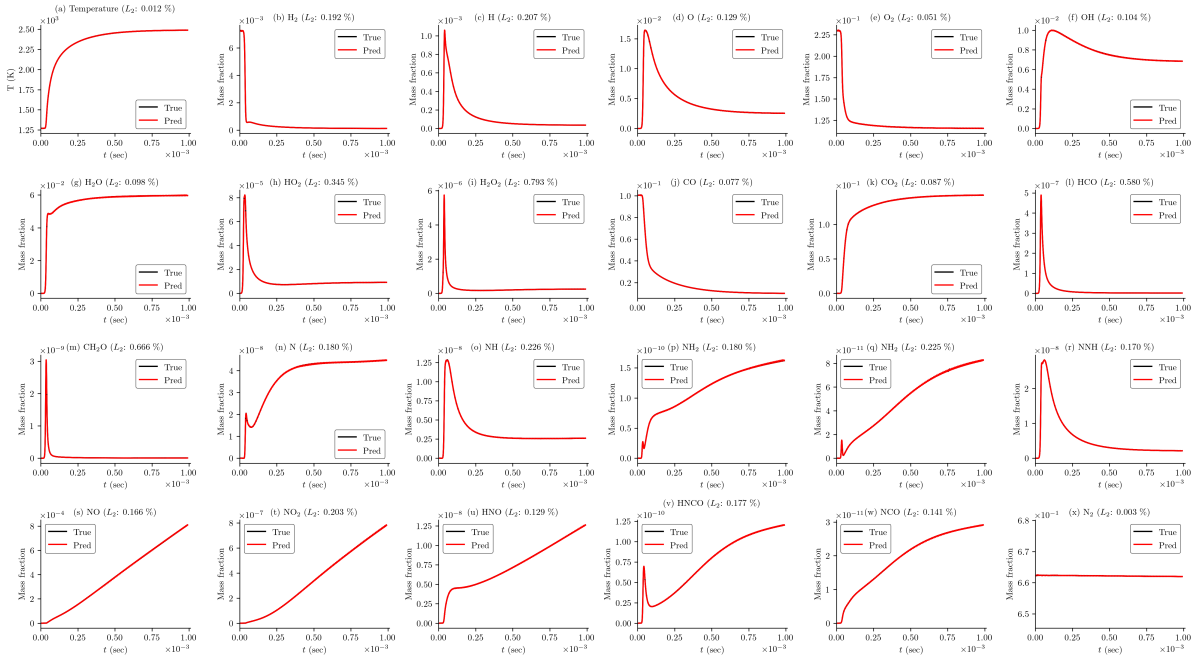


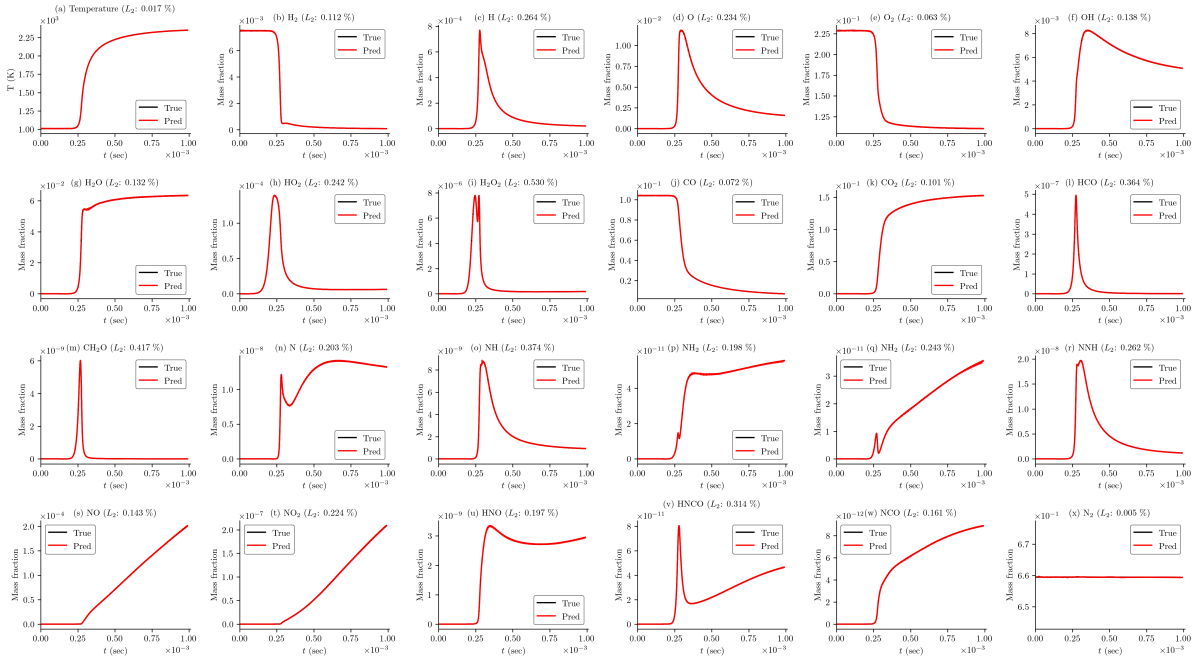
Fig. C.10: **GRI problem, convergence of DeepOKAN.** Convergence of error for GRI problem for trunk training and branch training when trained using two-step training of DeepOKAN. Convergence of mean of relative L_2 error in trunk and branch training with epoch for adaptive loss Type-B. The plots show the convergence for the three independent runs. The three runs follow a similar convergence path with epochs. We have also shown the learning rate considered for each method. We have considered 20 mini-batch up to 6 k epoch and 40 mini-batch after 6 k epoch during the trunk training and indicated by the vertical line at 6 k epoch. Similarly, We have considered 60 mini-batch up to 5 k epoch and 120 mini-batch after 5 k epoch during the branch training and indicated by the vertical line at 5 k epoch

Table C.6: **GRI problem, mean and standard deviation** of the % relative L_2 error for the test samples. The state variable describing the dynamics of H_2O_2 has the maximum mean relative L_2 error. The violin plots for the relative L_2 error are shown in Fig. 11. The relative L_2 errors are calculated similarly as discussed in Section 4.3.

States	μ	σ	States	μ	σ	States	μ	σ
Temp	0.008	0.002	H_2O_2	0.362	0.095	NH_2	0.117	0.028
H_2	0.100	0.024	CO	0.037	0.014	NNH	0.094	0.028
H	0.121	0.034	CO_2	0.059	0.012	NO	0.095	0.023
O	0.096	0.022	HCO	0.209	0.064	NO_2	0.127	0.029
O_2	0.057	0.019	CH_2O	0.284	0.104	HNO	0.099	0.025
OH	0.064	0.014	N	0.101	0.023	HNCO	0.100	0.034
H_2O	0.068	0.014	NH	0.102	0.045	NCO	0.089	0.023
HO_2	0.240	0.077	NH_2	0.096	0.026	N_2	0.003	0.001



(a) Sample 2



(b) Sample 3

Fig. C.11: **GRI problem, sample result from test dataset.** Plot showing the two sample predicted dynamics of the state variables for GRI problem. The sample result corresponds to one of the higher errors in prediction. The predicted sample shows good agreement with the true value.

C.3 Implementation and additional result for AMORE in FNO

FNO [31] consists of multiple Fourier layers. In Fig. C.12, we have shown a schematic of a FNO which maps input function to a output function $a(x) \mapsto u(x)$. Each Fourier layers consists of 3 parts: an FFT, which transforms the input function to the frequency domain, and then the selected modes are multiplied by learnable parameters. The second part is an inverse FFT which transforms the weighted modes to physical domain. The third part is a weighted (learnable) residual connection added to the inverse FFT output. After the residual connection an activation is considered. There are also two network at the beginning and end of FNO which uplift and down-lift the input and output respectively.

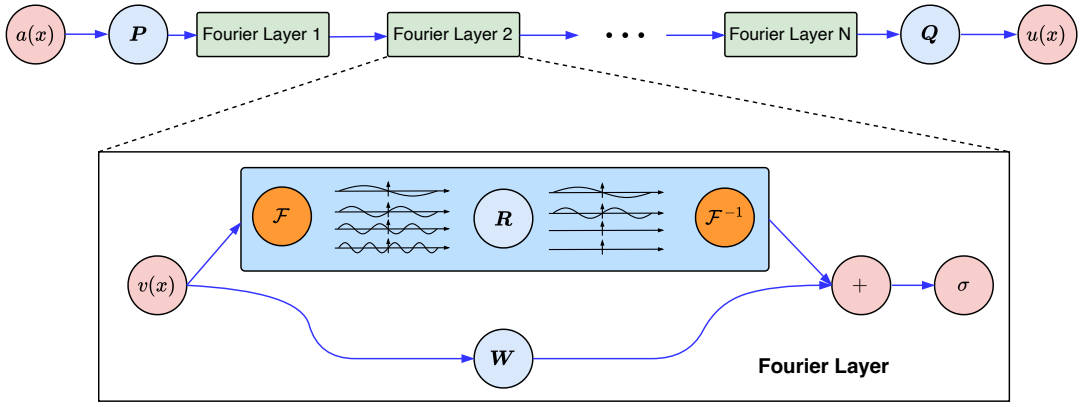


Fig. C.12: **Schematic of a FNO.** In the top, we have shown a complete FNO which includes (i) lift to a higher dimensional channel space using a network $\boldsymbol{P}$ (e.g., a single layer without an activation function), (ii) a series of Fourier layers and (iii) a projection network $\boldsymbol{Q}$ (e.g., a single layer without an activation incitation). In the bottom, we have shown a detail of a Fourier layer. Inside a Fourier layer, the input function is transomed to frequency domain using FFT, the selective modes are multiply with learnable parameters. The weighted modes are transformed to physical domain using IFFT. This output is added to a weighted (learnable) residual connection. After the residual connection an activation is considered. The trainable parameters are the uplift network $\boldsymbol{P}$, the projection network $\boldsymbol{Q}$, the weights in the Fourier domain $\boldsymbol{R}$ and the residual network $\boldsymbol{W}$.

Similar to the DeepONet problem, we consider the same time decomposition of $n_t + 1 = 101\Delta t$ and map the initial condition of each segment to the time history of 101 time points. Normalization scheme considered for the input initial condition and the output is similar to the one considered for the syngas problem. The time points are normalized to $[-1, 1]$ using the minimum and maximum value of t at t_0 and t_{101}. The time coordinates are concatenated with the feature dimensions thereby making the input size $bs \times (n_t + 1) \times 13$. The output size $bs \times (n_t + 1) \times 12$. In the present study, we consider five Fourier layers with 64 hidden dimension and 16 modes retain after the FFT. The gelu activation function is considered for the Fourier layer except for the last Fourier layer. The total number of trainable parameters is 350156. As discussed in Section 6, we consider two loss functions, the first one is the non adaptive loss function (FNO-NA) given by Eq. (7) and second one is the adaptive loss function Type-B (FNO-Ad-B) given by Eq. (14). We also note that in this case, we have not considered the loss due to conservation of mass and the output bound considered in DeepONet given by Eq. (32). We train out FNO model using Adam optimizer in a PyTorch environment with 64-bit data precision. We updated the adaptive weights at every 10 epoch after 10^{th} epoch. The mean relative L_2 error in training and testing for three different runs are shown in Table C.7 and the convergence of error with epoch are shown in Fig. C.13. Similar to the DeepONet, as discussed in Section 4.3, we study the results with the test dataset in more detail. The mean and the standard deviation of the relative L_2 error of the reconstructed test dataset for each state variable for both the method for FON-NA and FNO-Ad-B are shown in Table C.8. The violin plots of the relative L_2 error for of the state variable for FON-NA and FNO-Ad-B are shown in Fig. 13. We observed that the

Table C.7: **Training and testing error for FNO, Syngas problem.** Relative L_2 error in training and testing for FNO when trained with different loss functions. In the second column, w e have indicated the type of loss function considered for each case. The third and fourth columns show the mean relative L_2 error (%) for training and testing, respectively, for three independent training runs for each case. The convergence of the mean of relative L_2 error with epoch is shown in Fig. C.13. The last column shows the training time for three different runs. We also like to mention that the training time include the calculation of relative L_2 error in both norm and physical space at every 10 epoch.

Case	Loss function	Relative L_2 error (%)		Training time (Hrs)
		Training	Testing	
FNO-NA	Non-Adaptive	0.167, 0.173, 0.170	0.166, 0.172, 0.169	13.72, 13.69, 13.77
FNO-Ad-B	Adaptive: Type-B	0.126, 0.128, 0.128	0.125, 0.128, 0.128	14.66, 14.8, 14.61

Table C.8: **Syngas problem, FNO, mean and standard deviation of the % relative L_2 error** for the reconstructed test samples when predicted using FNO, i.e., FNO-NA and FNO-Ad-B. The violin plots of the relative L_2 error are shown in Fig. 13. The relative L_2 errors are calculated similarly to those discussed in Section 4.3.

Case		T	H_2	O_2	O	OH	H_2O	HO_2	H	CO	CO_2	HCO	N_2
FNO-NA	μ	0.015	0.199	0.082	0.373	0.184	0.268	0.598	1.395	0.052	0.159	0.941	0.003
	σ	0.004	0.083	0.026	0.093	0.047	0.067	0.155	0.261	0.014	0.048	0.214	0.001
FNO-Ad-B	μ	0.024	0.158	0.084	0.272	0.147	0.214	0.347	0.608	0.064	0.137	0.519	0.008
	σ	0.005	0.032	0.017	0.049	0.030	0.047	0.066	0.080	0.014	0.032	0.096	0.002

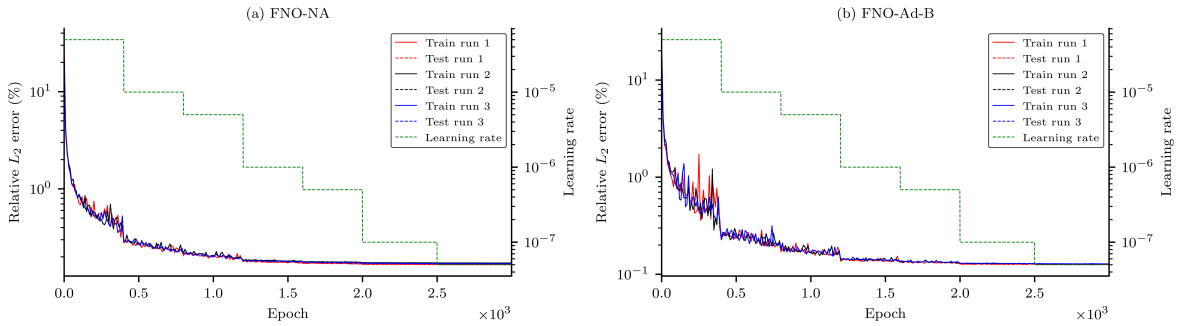


Fig. C.13: **Syngas problem, convergence of FNO.** Convergence of mean of relative L_2 error in training and testing with epoch for FNO and different loss function (a) FNO-NA (b) FNO-Ad-B. The plots show the convergence over the three independent runs. The three runs follow a similar convergence path with epochs. We have also shown the learning rate for one of the runs. We consider a batch size of 286 (720 mini-batches) for training.

D Additional discussion on Computational cost

In the present study, we have considered two different types of DeepONet, i.e., DON, where both branch and trunk are ResNet, and DOK, where the branch is ResNet and the trunk is KAN, and two different paradigms of training methods. In the previous sections, we have discussed the errors for each of the methods for two different problems, i.e., the syngas and the GRI-Mech 3.0. The training time for each of the methods and three independent runs is shown in Table D.1. The DeepONets are trained using NVIDIA L40S GPU (https://www.nvidia.com/en-us/data-center/l40s/) in a network cluster. The variations in computation time are different even for the same method due to network latency and the existing state of the computing and node sharing. The training time for adaptive loss functions is marginally higher. Furthermore, the computational time for the two-step method is higher than the one-step method. In the case of trunk training of the two-step training, we need to train an additional tensor ($\boldsymbol{A}$) whose size is too large. Future research may focus on reducing the computational time for the two-step method. We also observed that the training time for the trunk in the case of the GRI problem does not scale with the number of parameters, number of state variables, and number of epochs considered compared to the syngas problem, even though the number of parameters and dataset size are larger. This is because we use a smaller number of mini-batches in the training of the trunk for the GRI problem compared to the syngas problem.

Table D.1: **Training time for Syngas and GRI problem** for different method considered. We observed that the training time for the adaptive loss functions are marginally higher compared to the non-adaptive loss function. The training time for two-step training is higher than one-step training. We also observed that the training time is marginally different even for the same method. This may be due to the network latency and the existing state of the computing and node sharing. We trained the DeepONets using NVIDIA L40S GPU using Adam [32] optimizer in Tensorflow version 2 environment with 64-bit data precision. We also like to mention that we have calculated the adaptive weights in norm and physical space using mini-batch and numpy after every 50 epochs. Similarly, we have calculated the relative L_2 after every 50 epochs in norm and physical space using mini-batch and numpy. Since we have calculated L_2 error it added approximate 40 minutes time for L_2 error calculated.

Syngas problem			
Training time: One-step training			
	Case	Loss function	Time (hrs)
DeepOnet for syngas problem (15 k epochs)	DON-NA	Non-Adaptive	10.91, 10.32, 10.86
	DON-Ad-A	Adaptive: Type-A	11.42, 11.37, 11.41
	DON-Ad-B	Adaptive: Type-B	11.24, 11.05, 11.28
DeepOKAN for syngas problem (15 k epochs)	DOK-NA	Non-Adaptive	10.59, 10.56, 10.45
	DOK-Ad-A	Adaptive: Type-A	11.31, 11.71, 11.80
	DOK-Ad-B	Adaptive: Type-B	11.72, 11.71, 11.55
Training time: Two-step training			
Training time: trunk training			
	Case	Loss function	Time (hrs)
Trunk (ResNet) training for syngas problem (15 k epochs)	ResNet-NA	Non-Adaptive	19.28, 19.33, 19.36
	ResNet-Ad-A	Adaptive: Type-A	19.83, 19.88, 20.25
	ResNet-Ad-B	Adaptive: Type-B	20.58, 20.70, 20.82
Trunk (KAN) training for syngas problem (15 k epochs)	KAN-NA	Non-Adaptive	19.14, 19.54, 19.13
	KAN-Ad-A	Adaptive: Type-A	19.68, 19.69, 20.49
	KAN-Ad-B	Adaptive: Type-B	19.61, 19.89, 19.71
Training time: branch training			
	Case	Loss function	Time (hrs)
Branch (ResNet) training for syngas problem	2S-NA	Non-adaptive	9.34, 9.84, 9.89
	2S-Ad-A	Adaptive: Type-A	9.90, 10.34, 10.38
	2S-Ad-B	Adaptive: Type-B	10.32, 9.89, 10.30
GRI-Mech 3.0 problem			
Training time: Two-step training			
Training time: trunk training			
	Case	Loss function	Time (hrs)
Trunk (KAN) training for GRI problem (20 k epochs)	KAN-Ad-B	Adaptive: Type-B	23.19, 23.25, 23.42
Training time: branch training			
	Case	Loss function	Time (hrs)
Branch (ResNet) training for syngas problem (20 k epochs)	2S-Ad-B	Adaptive: Type-B	25.78, 25.36, 25.47